UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Creación de entorno seguro en servidor cloud y entorno bare-hardware

Isaac Alejandro Serrano González
Tutor: Jorge Carretie Romero
Ponente: Javier Aracil Rico

MAYO 2019

CREACIÓN DE ENTORNO SEGURO EN SERVIDOR CLOUD Y ENTORNO BARE-HARDWARE

AUTOR: Isaac Alejandro Serrano González
TUTOR: Jorge Carretie Romero

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2019

Resumen (castellano)

Durante mucho tiempo, la estructura básica necesaria para poner en funcionamiento una página web consistía en un servidor remoto donde alojábamos los ficheros, el cuál exponía nuestras aplicaciones al público. Normalmente una página web se encontraría detrás de un servidor web, como Apache o Nginx, junto con otros servicios, como bases de datos, que podrían estar o no presentes en la misma máquina

Pero con el tiempo esta estructura se ha ido quedando cada vez más obsoleta, presentando una serie de problemas que empañan la experiencia del cliente, como por ejemplo: falta de capacidad para satisfacer la demanda del servicio, tiempos de no disponibilidad debido a la caída del servidor o errores de programación, vulnerabilidad ante los ataques al exponer partes críticas de nuestro sistema al exterior, etcétera.

Muchos de estos problemas pueden ser mitigados gracias a los avances en la contenerización y en los entornos distribuidos, los cuales usaremos para proponer una nueva forma de ofrecer nuestros servicios a los clientes.

Con la ayuda de la contenerización podemos limitar el número de elementos en nuestro sistema, utilizando solo lo necesario para mantener nuestra página en funcionamiento. De esta manera, reducimos la presencia de elementos innecesarios en nuestra máquina, aumentando los recursos disponibles del sistema y limitando el número de herramientas a las que un atacante tendrá acceso, a la hora de intentar manipular nuestra aplicación.

Finalmente, con el uso de los sistemas distribuidos somos capaces de separar los componentes de nuestro sistema, permitiendo un aislamiento que se asegura de limitar los daños o caídas del servicio a elementos individuales, que luego pueden ser sustituidos rápidamente gracias a los contenedores que forman el sistema de Kubernetes.

Abstract (English)

It has been a long time since there was a significant change in the way we provide our services to the public, dragging classic errors that, in one way or another, end up worsening the experience of our customers. We are talking about classic errors such long downtimes due to failures in our server that not only bring down the main service in our infrastructure, it also hits our secondary services that may or may not depend (or are even involved) in that service, going down either way and leaving us with multiple services unavailable just because one of them suffered a failure.

However, thanks to advances in distributed computing and the increasing popularity in containerized applications we are able to bring to the table a new way to organize our infrastructure that will improve the means we use to bring our services to the public.

Thanks to containerized applications we are able to minimize the number of elements in our system, keeping only the essentials in order to expose our services working and limiting the tools that an attacker can use in order to access our system and manipulating it.

Finally thanks to distributed computing we are able to keep the components of our system in different computing resources allowing us to keep our resources isolated in order to limit the amount of damage in case one of them starts failing, and also being capable to scale up components of our architecture in case we observe an increase in the use of our system; all of this in the palm of our hands thanks to the Kubernetes application.

Palabras clave (castellano)

Contenedor, nodo, nube, pod, sistemas distribuidos

Keywords (inglés)

Cloud, container, distributed computing, node, pod

Agradecimientos

Quiero agradecer a mis padres su apoyo durante la realización de mi trabajo y a Ezenit por darme la oportunidad de realizar un proyecto que por mis propios medios habría sido imposible de realizar.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

Durante algún tiempo, la estructura que utilizamos para servir páginas presenta problemas evidentes. Cualquier fallo en la infraestructura de uno de sus componentes puede provocar la caída de todo el sistema, el cual requiere una intervención manual por nuestra parte, para conseguir que vuelva a su correcto funcionamiento. Además de que, al tratarse de sistemas compartidos entre varias páginas o servicios, nos arriesgamos a que el fallo de un sistema no directamente relacionado con nuestro programa pueda provocar un daño colateral en otros servicios, afectando a más de una de nuestras páginas.

Con la evidente presencia de problemas a la hora de montar nuestro sistema web, hemos decidido buscar una alternativa en la arquitectura que nos permita responder mejor a la caída de nuestros sistemas, al mismo tiempo que los protegemos ante ataques de denegación de servicio (o incrementos naturales del tráfico en la web) y fallos en los elementos individuales de nuestro sistema.
Para implementar esta arquitectura hemos decidido utilizar la tecnología de los sistemas distribuidos que ofrece Kubernetes.

1.2 Objetivos

El objetivo de este TFG es crear una nueva estructura para el despliegue de páginas web, que cumpla las siguientes características:

- Crear un entorno de hosting renovado, que no muestre diferencias aparentes a la vista del cliente, con tiempos de repuestas iguales o mejores a los que se tenían con arquitecturas previas.
- Utilización de frameworks, que eviten ataques más básicos, como la inyección SQL o similares.
- Ser capaces de monitorizar el tráfico que llega a nuestro sistema, para captar errores y diagnosticar fallos de funcionamiento
- Recibir alertas ante caídas de algunos elementos de nuestro sistema
- Escalado automático de los recursos para satisfacer picos en el tráfico de nuestra página y mitigar denegaciones de servicio
- Crear un plan de integración continua, que permita actualizar nuestro sistema, evitando largas caídas de nuestro servicio, al mismo tiempo que implementamos medidas de rollback en caso de un fallo en la implementación de la actualización
- Copias de seguridad automáticas y con protección de acceso

1.3 Organización de la memoria

- **Estado del arte:** En esta sección describiremos la tecnología que se va utilizar para llevar a cabo este proyecto, al mismo tiempo que describimos tecnologías similares en el mismo ámbito, que podrían haberse utilizado para llevar a cabo nuestro proyecto.
- **Diseño:** En esta parte de la memoria hablaremos sobre cómo hemos organizado la arquitectura de nuestro sistema de hosting, empezando por las partes más básicas de nuestro sistema (los contenedores) y escalando a los conceptos más avanzados de los sistemas distribuidos, donde alojaremos nuestros contenedores (componentes de Kubernetes). Según vayamos describiendo los elementos, iremos comentando las

medidas de seguridad que vamos a ir implementando en algunos de los lugares más críticos.

- **Desarrollo:** Utilizaremos este bloque para describir como hemos implementado las soluciones para cada uno de los aspectos de la arquitectura definidos en la sección del diseño. También iremos describiendo alternativas que se plantearon durante el desarrollo y las dificultades a la hora de implementar algunas de las medidas que se propusieron como objetivo, a la hora de iniciar el proyecto.
- **Integración, pruebas y resultados:** Para finalizar la parte del desarrollo, mostraremos los resultados de nuestro trabajo comprobando que objetivos hemos sido capaces de cumplir y mostraremos algunos test que haremos sobre el sistema, como la medición de tiempos.
- **Conclusiones y trabajo futuro:** En esta última sección, analizaremos el trabajo realizado y propondremos estudios para mejorar la rentabilidad de la nueva infraestructura.

Durante el transcurso de la redacción de esta memoria podremos diferenciar 2 pilares básicos en nuestra aplicación: la contenerización (Docker) y los sistemas distribuidos (Kubernetes); los cuales tendrán medidas y procedimientos especiales para añadir capas de seguridad extra en nuestra arquitectura.

2 Estado del arte

2.1 Contenerización

Todos hemos oído hablar de las máquinas virtuales, con las cuales, gracias al uso de software, somos capaces de replicar el comportamiento de otro sistema desde el nuestro. El uso de este software nos permite limitar recursos y probar aplicaciones dentro de un entorno aislado, sabiendo el comportamiento que nuestra aplicación tomará de antemano, sin que afecte al sistema que aloja la máquina virtual (VM).

La contenerización sigue un concepto de aislamiento parecido, pero con la diferencia de que evitamos la carga de mantener una máquina virtual entera por aplicación llevándonos solo elementos fundamentales del kernel de Linux y ejecutando todos los contenedores en un mismo entorno, creando imágenes de nuestras aplicaciones más pequeñas y compactas, que reducen la capacidad necesaria para ejecutarlas y limitan los elementos a los que un atacante tendrá acceso para intervenir en nuestro sistema.

Algunas de las ventajas que podemos mencionar sobre las aplicaciones contenerizadas son la capacidad de realizar este proceso sobre casi cualquier aplicación, su portabilidad (al ser entornos aislados no dependen del sistema donde se alojan), la capacidad de realizar actualizaciones de forma continua y la facilidad para escalar este tipo de aplicaciones.

https://searchdatacenter.techtarget.com/es/definicion/Contenerizacion-de-aplicacion-contenerizacion-de-app [1]

2.1.1 Docker

Docker es un programa que aprovecha la tecnología de contenedores para crear imágenes, que contienen una versión compactada de un sistema, que contiene los mínimos elementos de un sistema operativo, más la aplicación que queremos compactar. Mientras que una máquina virtual necesitaría más elementos del sistema operativo, que no son necesarios para nuestra aplicación, una imagen de Docker arranca un proceso que solo usa la memoria del ejecutable y que arranca en un contenedor que funciona nativamente en Linux (pero no contiene todos los elementos que una máquina virtual con sistema operativo Linux instalado necesitaría).

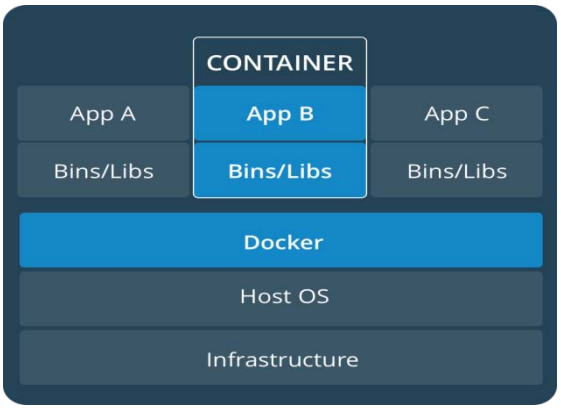

Figura 2-1-1-1: Esquema docker

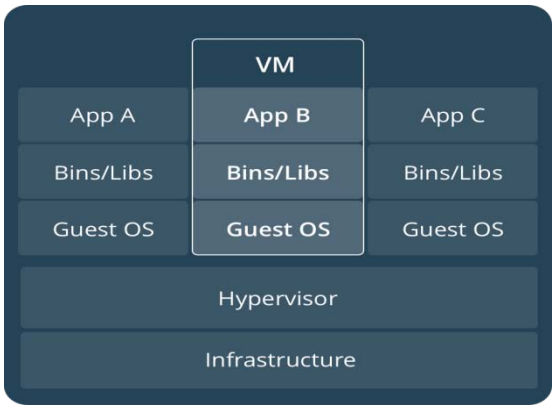

Figura 2-1-1-2: Esquema VM

https://www.stackfire.com/docker-que-es-y-como-funciona-la-contenerizacion/ [2]

2.1.2 Alternativas

Rkt (en minúsculas) es una aplicación que también trabaja a nivel de contenedores y que funciona con entornos cloud a nivel de producción, haciéndolo compatible con otras aplicaciones como kubernetes, las cuales necesitaremos para crear nuestra arquitectura.

SingularityContainer podría ser otra alternativa a Docker, con su capacidad de crear aplicaciones contenerizadas, que además de ser compatible con herramientas de la nube, presume de ser capaz de aguantar grandes cargas de trabajo, lo cual es común en el ámbito del aprendizaje automático y las analíticas de predicción.

https://coreos.com/rkt/ [3]
https://www.sylabs.io/singularity/ [4]
https://containerjournal.com/2019/01/22/5-container-alternatives-to-docker/ [5]

2.2 Sistemas distribuidos

Como se puede inferir por el nombre, los sistemas distribuidos son la separación de los componentes de nuestra arquitectura en distintas máquinas comunicadas entre sí a través de una red de comunicación, de tal manera, que un usuario no sea capaz de notar la diferencia si hubiésemos alojado nuestra aplicación en un sistema centralizado.

Sin embargo, alguna de las ventajas que se pueden sacar a un sistema distribuido es la independencia de sus componentes, haciendo que la caída de uno de ellos no afecte directamente a los otros y la escalabilidad de este sistema, permitiendo responder a una mayor demanda del servicio y, al mismo tiempo, mantener la continuidad del servicio si uno de los componentes replicados ha dejado de funcionar.

De todas maneras, no será la arquitectura del propio sistema distribuido (que puede variar enormemente de una arquitectura a otra) en lo que nos centraremos en esta parte de la memoria, sino que buscamos una herramienta que sea capaz de gestionar nuestras aplicaciones en un sistema distribuido, encargándose de desplegar nuestra aplicación en cada uno de los nodos de computación, actualizar dichos componentes cada vez que hagamos una actualización en nuestra aplicación o uno de los nodos no pueda dar servicio y escalar nuestra aplicación en caso de que haya un incremento en el uso de nuestros recursos.

https://www.universidadviu.es/sistemas-distribuidos-caracteristicas-clasificacion/ [6]

2.2.1 Kubernetes

Kubernetes es una aplicación con la capacidad de organizar y exponer aplicaciones contenerizadas al exterior, gracias a la cual podemos dejar la carga de trabajo sobre cómo organizar nuestras aplicaciones en el cluster a Kubernetes, ya que, gracias al uso de kubelet y cluster-autoscaler, esta aplicación recogerá estadísticas sobre el uso de los recursos de la plataforma y decidirá donde alojar cada aplicación, abstrayendo la organización de los elementos del programador.

Además, toda la parte de la organización de la red de comunicaciones que comunicará los diferentes elementos internos del cluster (resolver dominios e IPs internas, por ejemplo) quedará también en manos de la aplicación, dejando para nosotros la definición de algunos

elementos puntuales (servicios, balanceadores, etc.) que queremos que se usen para organizar nuestras aplicaciones.

Las ventajas que se obtienen al usar esta herramienta, en vez de montar nuestro cluster a mano, es la facilidad que tenemos para escalar los elementos de nuestro sistema (incluso automatizarlos), la capacidad de realizar actualizaciones sin que el servicio se vea interrumpido, la capacidad de observar el estado de los recursos de todo el cluster con el uso de un mismo grupo de comandos (kubectl), la portabilidad de los componentes (somos capaces de llevarlos a un entorno cloud distinto o bare-hardware, sin notar apenas ninguna diferencia) y la capacidad de utilizar nuestros recursos eficientemente, dejando que el cluster reduzca sus recursos en momentos de menor uso, reduciendo los costes necesarios para mantener la infraestructura.

Figura 2-2-1-1: Logo kubernetes

https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#what-does-kubernetes-mean-k8s [7]

2.2.2 Alternativas

Una de las tecnologías que hemos visto previamente es Docker, la cual nos permite crear aplicaciones dentro de contenedores y puede funcionar junto a Kubernetes, pero no depende necesariamente de ella. Si indagamos un poco en su documentación, podemos encontrar la herramienta DockerSwarm que, de manera similar a un cluster de kubernetes, crea un cluster usando máquinas virtuales, como los nodos que alojarán nuestros contenedores. La máquina virtual inicial funciona como máquina maestra, mientras que las nuevas máquinas que se unan al cluster funcionarán como nodos trabajadores. Con la ayuda de un fichero YAML (similar a los de kubernetes), llamado docker-compose, somos capaces de crear contenedores con especificaciones especiales sobre su presencia en el cluster (como el número de replicas), que se repartirán entre los nodos de nuestra arquitectura de manera similar a lo que harían si fuesen lanzadas con la herramienta Kubernetes (incluso se añaden balanceadores de carga).

Otra alternativa podría ser Apache Marathon que, según su página web, también es una aplicación que funciona como organizador de aplicaciones contenerizadas y comparte muchas de las características que buscamos en un organizador: distribuye aplicaciones contenerizadas, comprueba el estado de nuestros sistemas y contenedores, permite escalarlos con relativa facilidad y nos devuelve métricas de uso en un formato fácil de entender y analizar (JSON).

https://docs.docker.com/get-started/part4/ [8]
https://mesosphere.github.io/marathon/ [9]
http://techgenix.com/kubernetes-alternatives/ [10]

2.3 Servicios en la nube

A día de hoy, la mayoría de los servicios en la red que utilizamos podrían definirse como computación en la nube, ya que normalmente el servicio no se encuentra físicamente en nuestro sistema, sino que accedemos a él a través de una aplicación, que nos conecta y obtiene la información a través de la red, conectándose a un proveedor de servicios que tiene alojada la aplicación.

Gracias al esquema anterior somos capaces de tener una gran variedad de servicios a nuestra disposición, siempre y cuando tengamos conexión a internet, permitiendo que nuestro equipo acceda a webs, tiendas, almacenamiento de datos, etc. Es como si estuviera todo en nuestro equipo.

En lo referente a nuestro proyecto, nos interesa la nube (o más concretamente un proveedor), ya que necesitamos una sola fuente que sea capaz de suministrarnos los recursos necesarios para montar nuestra infraestructura (almacenamiento, cache, servidores, dominios, etc.), por eso vamos a hacer uso de proveedores de servicios, que no solo nos darán acceso a aplicaciones que nos permitan monitorizar nuestros recursos, sino que también (a cambio de una cuota mensual) se encarguen de la gestión y suministro de los recursos que vamos a necesitar para tener nuestra aplicación en línea

https://azure.microsoft.com/es-es/overview/what-is-a-cloud-provider/ [11]
https://geekland.eu/que-son-los-servicios-en-la-nube/ [12]

2.3.1 Amazon Web Services

Amazon Web Services es un proveedor de servicios en la nube, propiedad de Amazon, que utilizaremos para la adquisición de arquitectura necesaria para poner en funcionamiento nuestros sistemas.

Gracias al uso de la nube, nos ahorramos las dificultades que tendríamos a la hora de reservar un servidor, pudiendo utilizar las instancias EC2 para obtener unidades de computación, que servirán para alojar nuestros contenedores. Además, disponemos de AWS S3, que funciona como almacenamiento de ficheros en la nube y que pueden ser utilizados a modo de volumen (esto enlazará con la coherencia del sistema cuando hagamos componentes redundantes y en paralelo). También contaremos con volúmenes tradicionales que, a diferencia de S3, no tienen problemas de velocidad, debido a temas de sincronización. La lista sigue con elementos como Cloudwatch, que sirve para monitorizar nuestros sistemas, lo cual ayuda a calcular costes y aumenta la seguridad de nuestro sistema, con el uso de alertas a componentes monitorizados.

Podemos seguir encontrando características que añadir a nuestro sistema (como CDN para la propagación de estáticos de nuestra página), ya qu,e otras de las ventajas de estos proveedores en la nube, es la capacidad de añadir nuevas aplicaciones o funcionalidades mientras pasa el tiempo, sin que afecte a las características que ya estamos usando; todo esto con un coste que se va ajustando al crecimiento de nuestro negocio y nos permite crecer a la par que las necesidades de nuestros clientes.

Figura 2-3-1-1: Logo AWS

https://aws.amazon.com/es/about-aws/ [13]
https://aws.amazon.com/es/products/?nc2=h_m1 [14]

2.3.2 Alternativas

Actualmente podemos encontrar una gran variedad de páginas que ofrecen un tipo u otro de servicio en la red, que podría entrar dentro de la categoría que estamos describiendo y que pueden satisfacer algunos de los requisitos que buscamos. Podríamos usar aplicaciones mundialmente conocidas como Dropbox o Google Drive para el almacenamiento de cualquier tipo de fichero fuera de nuestro sistema.

Sin embargo, si buscamos un proveedor de servicios similar a AWS, podríamos usar de alternativa Microsoft Azure para satisfacer algunas de las necesidades clave para el desarrollo de nuestra arquitectura, concretamente podríamos usar AzureKubernetesService para crear el servicio, que se encargaría de orquestar nuestros contenedores de manera más simplificada, o Azure SQL Database que nos permite tener una base de datos como servicio a la que nos podemos conectar desde nuestra aplicación (la cual no estaría en el mismo lugar y aislando de esta manera ambos sistemas).

Otra alternativa, más centrada en el entorno Kubernetes específicamente, sería DigitalOcean, que parece más centrada en la creación y mantenimiento de clusters centrados en aplicaciones contenerizadas (suministran máquinas virtuales que funcionarían de manera similar a las instancias EC2), las cuales pueden haber sido generadas por diversos medios, ya sea Docker, Cassandra u otros.

https://azure.microsoft.com/es-es/overview/what-is-azure/ [15]
https://azure.microsoft.com/es-es/services/kubernetes-service/ [16]
https://www.digitalocean.com/ [17]

3 Diseño

3.1 Contenedores

Los contenedores forman la parte más básica de los componentes de nuestro sistema, cada uno de ellos debe contener una aplicación principal, que representa la funcionalidad que ocupan en el cluster (dígase base de datos, apache, etc.).

La aplicación que arranca dentro de un contenedor se denomina imagen y es la que contiene la lista de instrucciones que han creado nuestro entorno dentro del contenedor.
A la hora de crear una imagen en Docker, tenemos que crear un fichero llamado Dockerfile, el cual contendrá la lista de comandos necesarios para instalar una imagen, que será arrancada en una máquina virtual. También existe un fichero llamado docker-compose, que permite definir una estructura más compleja de conjuntos de contenedores, que pueden interactuar entre sí, pero eso queda fuera del proyecto ya que no es compatible y necesario con requisitos que examinaremos más adelante.

Dockerfile tiene su propia estructura de comandos, y algunos de los que usaremos son los siguientes:

- **FROM**: Este comando es la base de todo el fichero. Con esta línea, definimos la imagen en la que se basará nuestro contenedor, ya sean imágenes creadas por nosotros en local o imágenes subidas en repositorios en la nube (oficiales de Docker Hub o repositorios ajenos). Básicamente, puedes crear imágenes sucesivas que hereden unas de otras, cada una con los componentes necesarios para iniciar la aplicación que engloban. Por poner un ejemplo, podrías partir de la imagen de oficial de apache que encuentra en Docker Hub, lo cual generaría un servidor Apache en el que podríamos alojar nuestros ficheros; y luego en instrucciones sucesivas instamos PHP, permitiéndonos montar una página web con ficheros de extensión php, sin haber tocado nada de configuración de Apache en nuestro Dockerfile (es solo un ejemplo ilustrativo ya que ya existe una imagen oficial de PHP+Apache, pero su configuración en php.ini no está pensada para un entorno de producción).
- **WORKDIR**: Este comando nos permite definir el directorio donde lanzaremos los comandos sucesivos
- **ADD/COPY**: Gracias a estos comandos, somos capaces de añadir los ficheros de nuestro entorno al interior de la imagen, permitiéndonos añadir los ficheros que queremos que la imagen tenga presente, cada vez que se arranca un contenedor. Estos ficheros conservan los permisos con los que los copiamos, pero el grupo y su propietario pasan a ser del usuario interno de la imagen que esté ejecutando los comandos (por defecto sería root-root), pero podemos definir otro propietario usando el flag –chown, siempre y cuando exista dentro de la imagen (www-data:www-data para Apache, por ejemplo).
- **RUN**: Es el segundo comando central de un Dockerfile. Este comando sirve para arrancar instrucciones dentro de nuestra imagen, como si los estuviésemos lanzando dentro de una terminal, por lo que podemos utilizar este comando para una gran variedad de funciones. Hay que tener en cuenta algunos factores a la hora de usar este comando: no es interactivo, por lo que comandos que esperen respuesta por parte del usuario deben ser replanteados y que se ejecutan como el usuario de la imagen a no ser que se defina otro con –user.

- **ENTRYPOINT**: Con este comando, definimos la instrucción que se ejecutará una vez que la imagen este funcionando dentro de la máquina virtual (contenedor). Normalmente se utiliza este comando pasándole un script, que se ejecutará cada vez que se arranque la imagen, permitiéndonos realizar funciones que durante la creación de la imagen no funcionarían correctamente, como, por ejemplo, el arranque de servicios.
- **CMD**: Con este comando especificamos los argumentos que pasamos al entrypoint. Normalmente, cuando el entrypoint es un fichero, se suele añadir la línea exec "$@" al final del mismo; lo que hace esta línea es ejecutar los argumentos que se le han pasado al script, de tal manera que, si arrancamos la imagen con un httpd-foreground en una imagen de httpd de Docker Hub en el CMD, arrancaríamos apache al final del script.

  Una de las cosas que hay que tener en cuenta a la hora de arrancar una imagen es que es necesario mantenerla abierta con el uso de un comando que tenga su salida por terminal, ya que, si el proceso del entrypoint termina, el contenedor se cierra y perderíamos nuestro servicio. Esto se puede conseguir de varias maneras, ya sea lanzando nuestro proceso en modo foreground, redirigiendo uno de sus logs a la terminal, haciendo un tail –f de un log o de dev/null… Lo que sea necesario con tal de mantener la terminal abierta, pero teniendo en cuenta que esa salida funcionará como sistema de diagnósticos (logs) cuando lo subamos a nuestro entorno distribuido.

3.1.1 Servidor + Aplicación

Este contenedor es la clave de nuestras aplicaciones web que queremos exponer a nuestros clientes. No solo es necesario tener los ficheros de nuestras aplicaciones, sino que también necesitamos un programa que exponga esos ficheros al exterior.
La idea principal para esta imagen de Docker es alojar nuestros ficheros en un servicio que sea capaz de exponerlos al exterior y debe cumplir los siguientes requisitos:

- Tiene que ser efímero. Ningún fichero puede almacenarse dentro del pod si más tarde tiene que ser recuperado debido a la propia naturaleza de los contenedores, ya que si se arranca un contenedor tras haber sido terminado, los cambios previos se perderán.
- Tiene que funcionar en paralelo con pods iguales al mismo. Debido a la naturaleza de la replicación de pods, tenemos que estar seguros de que, al funcionar varios al mismo tiempo, no se producen problemas de consistencia en los datos.
- La información crítica (como los logs) tienen que mantenerse en un servicio externo

3.1.2 Sesiones

Normalmente, la sesión se almacenaría en el mismo lugar donde se encuentra nuestra página web y aunque esto es posible, como comentaremos en un futuro, también es probable que produzca fallos cuando intentemos combinar el uso de sesiones con la redundancia y disponibilidad. De momento, vamos a dejar de lado estas cuestiones y vamos a centrarnos en las características que se tienen que cumplir en nuestro contenedor para que sea posible llevarlo al cluster:

- Debe ser posible acceder desde fuera a nuestro contenedor. Con esto quiero decir que sea posible acceder desde el contenedor que contiene nuestra página web, que como veremos más adelante puede estar o no en la misma máquina.
- Tiene que tener una contraseña. Al ser posible acceder desde fuera, tenemos que asegurarnos de que exista algún tipo de método de autentificación, que nos permita diferenciar comunicaciones autorizadas o no.

3.1.3 Cache

Aunque, con tener una página web normal con suficiente capacidad para responder al tráfico debería bastar, podríamos tener un problema si la carga de tráfico se incrementa o si existen procesos que consumen una gran cantidad de recursos, ya que, aunque se mitiga con la replicación y uso de elementos en paralelo, el coste crecería de igual manera. Por eso, el uso de un contenedor con un sistema de cache para nuestra página puede aligerar la carga sobre los servidores de nuestra página.

Los requisitos son los mismos que los del contenedor de sesiones, a excepción de la contraseña, ya que la única comunicación que los contenedores deberían hacer a nuestro sistema de cacheado sería la limpieza de cache; aunque solo los contenedores asociados al cluster deberían ser capaces de hacerlo, por lo que pondremos un filtrado por ip, para saber quién tiene acceso a la limpieza de cache, mientras que el resto de usuarios simplemente se comunicarán con el contenedor para recibir una respuesta cacheada, o ser redirigidos a un contenedor de la página en caso contrario (técnicamente, la petición no cacheada también la devuelve nuestro sistema de cache, ya que lo ideal sería recuperar la petición y cachearla para usos futuros).

3.1.4 Base de datos

Los principios básicos para un contenedor que contenga nuestra base de datos son los siguientes:

- Debe existir un usuario con permisos necesarios para operar la base de datos, que otros contenedores puedan usar al conectarse de manera remota.
- Debe ser posible editar un fichero que contenga la configuración personalizada para la base de datos que se cargue cada vez que iniciemos el contenedor. La razón de usar este método, en vez de configurar la base de datos directamente en nuestra imagen, es para tener más versatilidad a la hora de crear bases de datos que se ajusten al tráfico y recursos que tenga la página (las conexiones máximas serían un buen ejemplo).
- Debe ser persistente. Debido a la restauración de estados, cada vez que un pod se reinicia es necesario que el estado de nuestra base de datos no se pierda.
- La base de datos solo debe ser creada la primera vez que se inicie el contenedor con nuestra imagen. En ejecuciones sucesivas, el contenedor cargará las bases de datos que existían previamente, antes de la creación del pod.

3.1.5 Medidas de seguridad implementadas

- Al usar Docker y sus contenedores, partimos de sistemas operativos con las mínimas herramientas instaladas, lo cual limita enormemente la cantidad de recursos a los que un atacante tiene acceso y evita que pueda usarlos para alterar el contenedor (las imágenes de Docker vienen sin elementos básicos, como editores de texto o comandos para hacer peticiones a la red).
- Como los contenedores son efímeros, no tenemos que preocuparnos por daños permanentes en nuestro sistema (la excepción sería la base de datos, pero hablaremos sobre las medidas tomadas en el siguiente apartado). Al seguir la filosofía de que los contenedores puedan perder sus estados sin consecuencias, podemos recuperarnos con facilidad cuando se produce un error; simplemente reiniciamos el pod y volvemos a un estado en el que nuestra página funcionaba con normalidad.
- Gracias a aislar nuestras aplicaciones en contenedores, si un atacante consigue acceder a una página, solo tendría acceso a los componentes de ese contenedor,

mientras que el resto de elementos estarían en otros contenedores, en los cuales el atacante no puede operar (incluso podría no llegar a saber de su existencia).

- Cuando vayamos a crear nuestra página web, haremos uso de frameworks que liberan al desarrollador de algunos de los aspectos más básicos de una tienda online (como el carrito), lo cual evita que se cometan fallos como la inyección SQL. Por poner un ejemplo, podemos hablar de Magento 2 y como el uso de funciones PHP nos permiten realizar consultas a la base de datos sin saber realmente como está estructurada, evitando el error clásico de crear un string en el que el atacante puede meter líneas SQL, que alteran el funcionamiento de la página (sin embargo, todavía existe la posibilidad de crear funciones específicas para acceder a la base de datos, por lo que habría que formar a los programadores de la existencia de estas funciones PHP antes de que implementen el código personalizado).
- Un factor interesante es que, al poner bases de datos en distintos contenedores, aunque el atacante se haga con el control de un usuario con superprivilegios, solo tendría acceso a la bases de datos de ese entorno, dejando las del resto de contenedores fuera de su alcance.
- Gracias al uso del comando COPY podemos generar una lista de ficheros para el entorno de desarrollo y otros para los de producción, de tal manera que podemos trabajar en nuestro equipo con configuración en modo local y luego automáticamente mover los ficheros de producción sin cambiar los originales y evitar llevar posibles errores a producción al olvidar volver a modificar los ficheros a su entorno original.
- Aprovechamos las herramientas básicas de Linux de grupos y permisos que vienen en todas las imágenes para controlar el acceso a los ficheros.
- Como es necesario mantener la "terminal" abierta para evitar que se cierre el contenedor podemos redirigir la salida de los logs de nuestra aplicación al stdout y al mismo tiempo evitar el log-flooding que ocurriría al escribir constantemente en un fichero (además luego con Kubernetes somos capaces de recuperar todos estas entradas agrupadas por entorno y acceder fácilmente a todos los logs del sistema).

3.2 Cluster

Kubernetes es una tecnología que nos permitirá distribuir nuestros contenedores y enlazarlos de manera correcta; una serie de conexiones dentro de un entorno privado que evita comunicaciones innecesarias con el exterior, que se encarga de distribuir el tráfico entre pods y que es capaz de volver a levantar sistemas caídos en caso de que uno de sus componentes deje de funcionar.

La cantidad de tipo de elementos que pueden aparecer en un entorno Kubernetes es muy variada; pero todos tienen en común que se pueden generar con un fichero yaml y que disponen de una serie de elementos obligatorios a definir para poder ser generados. Con el uso del comando kubectl explain podemos obtener una lista de parámetros para cada recurso, junto con sus descripciones. Algunos recursos que podemos añadir a nuestro cluster son:

1. **Pods**: Son la unidad básica de Kubernetes. Alojan uno o varios contenedores y siempre, cuando uno de ellos se mantenga arrancado (el PID padre sigue en funcionamiento), el pod se mantendrá en funcionamiento. Raramente utilizaremos un pod de manera individual, por lo que el siguiente grupo es el que definirá el elemento que alojará nuestras aplicaciones.
2. **Deployments**: Son conjuntos de pods administrados por un replicaset. Con el uso de deployments podemos gestionar el número de replicas de nuestros pods y, en combinación con otros elementos, podemos aumentar el número de replicas, según el

uso que hagamos de ellos. Es necesario usar labels para identificar a nuestros pods, de tal manera que el controlador es capaz de identificar a aquellos que debe monitorizar.

3. **Services**: Debido a que nuestros pods son efímeros y que pueden asignarse a cualquier máquina, según decida el balanceador de carga, es muy probable que nuestros pods cambien de IP, lo que hace que los pods que dependen de otros, necesiten tener las IPs de los pods que dependen actualizadas. Es aquí donde entran en juego los servicios, que en cierta manera funcionan de manera similar a los deployment, utilizando labels y sus selectores, para encontrar los pods que tienen que monitorizar y se encargan de servir de ruta para cualquier pod interno, que esté buscando otro elemento (en vez de poner una IP ponemos el nombre del servicio y nuestros pods utilizan el servicio para localizar uno de los pod bajo la vigilancia de este). Existen muchos tipos de servicio: ClusterIP que solo es visible dentro del cluster, NodePort que expone un rango de puertos al exterior usando la IP y puerto del nodo, LoadBalancer, que usa un balanceador de carga suministrado por nuestro proveedor en la nube y ExternalName que mapea un servicio a un dominio. De momento, usaremos ClusterIP, ya que no queremos que muchos de nuestros componentes se expongan al exterior y utilizaremos otro método cuando queramos exponer nuestro servicio al exterior.
4. **Ingress**: Este es el componente que utilizaremos para exponer nuestras páginas al exterior, exponiendo conexiones HTTP y HTTPS asignándolas a distintos dominios, que apuntarían a los servicios definidos previamente, siendo posible asignar subdominios de una misma página a distintos servicios, que apuntarán a todo tipo de pods.
5. **PersistentVolumes y PersistentVolumeClaims**: Estos dos componentes funcionan en conjunto para suministrar y controlar el acceso a los volúmenes que usarán nuestras bases de datos. El primero busca un volumen que cumpla ciertos requisitos especificados en nuestro yaml (tipo de almacenamiento y tamaño) y el segundo es la petición que realizará el pod para reservarse el acceso a ese volumen que ha sido creado por nuestro proveedor en la nube. Estos componentes pueden ser omitidos, ya que desde el propio deployment es posible especificar el volumen que queremos reservar, pero también pueden utilizarse para crear volúmenes de manera automática, para guardar datos que podríamos querer recuperar en caso de que se caiga el pod.
6. **Jobs**: Siguen el mismo principio que los pods, pero con una importante diferencia: están pensados para terminar tras cierto tiempo. La finalidad de este componente es lanzar tareas usando imágenes de los pods, con el objetivo de realizar una sola acción y luego cerrarse. Se pueden especificar el número de jobs paralelos y el número de veces que deben terminar con éxito, para considerar que la tarea se ha terminado; al mismo tiempo, son capaces de reiniciarse automáticamente si la tarea ha sufrido algún fallo un número limitado de veces. El problema con este recurso es que no se eliminan automáticamente por si mismos, ya que necesitan de un controlador para ser eliminados del cluster (como el que viene a continuación).
7. **Cronjobs**: Funcionan de manera similar a los cronjob que encontraríamos en un sistema Linux, pero haciendo uso de los Job para llevar a cabo tareas periódicas, según el plan determinado. Gracias a este componente, podemos ejecutar Jobs cada X tiempo y estar seguros de que el controlador se encargará de borrarlos y generarlos según se haya especificado, sin necesidad de intervención por parte del programador.
8. **HorizontalPodAutoscaler**: Este componente es capaz de leer métricas de uso de nuestros pods (uso de CPU, RAM,…) y establece el número de replicas de un deployment según los parámetros establecidos, permitiéndonos incrementar la capacidad cuando aumenta el tráfico en la página o cuando se realizan procesos costosos y queremos que la velocidad de la página no disminuya.

9. **Namespaces**: Aunque no es un recurso como tal, el uso de namespaces sirve para crear entornos aislados dentro de nuestro cluster que nos permiten separar los entornos de nuestras páginas y evitar que se pueda acceder a componentes que no estén relacionados con nuestro entorno a pesar de estar en el mismo cluster (o incluso nodo).
10. **Secrets**: Con este elemento podemos definir variables que contienen información sensible de nuestro cluster (contraseñas, direcciones, etc.) y pasárselas a nuestros componentes del cluster de tal manera que el recurso puede acceder a la información mientras que el resto de elementos solo saben que la variable existe, pero no pueden acceder a su contenido.

3.2.1 Medidas de seguridad implementadas

- Los pods son capaces de asignarse a cualquier tipo de nodo especificado, sin tener en cuenta el tipo de nodo que sea y suelen asignarse (si existen replicas) en nodos distintos, con el objetivo de que, si se cae un nodo, existe otro pod en otro, que sigue siendo capaz de mantener nuestro servicio.
- Los pods se vuelven a levantar si han caído, por lo que, si un atacante consigue tumbar nuestro servidor, el pod se reinicia a un estado previo y sigue funcionando con normalidad (deshaciendo los cambios que pueda haber hecho el atacante, siempre y cuando no fuesen a elementos persistentes).
- La posibilidad de lanzar tareas de manera periódica sin intervención, nos permiten crear backups cada cierto tiempo, haciendo posible recuperar estados de nuestros componentes, sin tener que hacer la copia nosotros mismos.
- El hecho de que muchos de nuestros componentes solo sean accesibles desde dentro del cluster, limita enormemente el acceso de un atacante a nuestro sistema, haciendo que solo pueda atacar directamente a los servicios que exponemos a la red, ya que el resto se encuentran aislados en otras máquinas dentro de nuestro cluster.
- El uso de elementos que escalan automáticamente según los límites que les especificamos, implica que podemos ajustar el uso de recursos según la demanda (disminuyendo el número de elementos reservados cuando hay poco tráfico) y podemos seguir dando servicio si sufrimos un pico en la demanda, al mismo tiempo que evitamos un crecimiento desmesurado si se tratase de una denegación de servicio (incluso podríamos seguir suministrando servicio a nuestros clientes, hasta cierto punto, gracias a los balanceadores de carga y la replicación, más de lo que podría hacer un servidor normal).
- Al usar distintos namespaces aislamos recursos que se pueden encontrar en la misma instancia (máquina virtual), pero que aún así no pertenecen al mismo conjunto de páginas, es decir, tenemos 2 páginas diferentes en la misma máquina pero ambas desconocen la existencia de la otra.

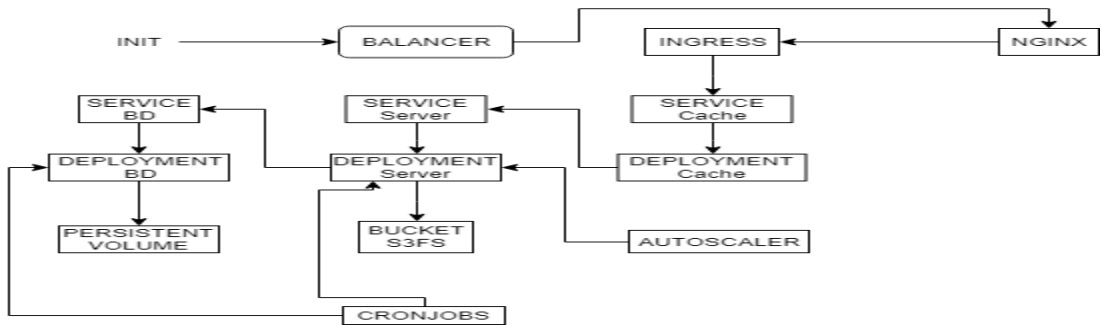

Figura 3-2-1-1: Esquema inicial namespace del cluster

4 Desarrollo

4.1 Creación de imágenes

Durante la creación de nuestras imágenes es importante tener en cuenta que la imagen debe realizar 2 procesos distintos, que llamaremos construcción y ejecución.
En la construcción debemos instalar todos los componentes necesarios para el funcionamiento de la imagen, al mismo tiempo que apuntamos a recursos de nuestro local para la creación de algunos ficheros que son necesarios para su correcto funcionamiento (concretamente Magento que necesita generar algunos ficheros conectándose a una base de datos), pero que antes de pasar a ejecución tienen que dejarse apuntando a los recursos que existirán en el cluster.
Durante la ejecución tenemos que pensar que aplicaciones deben iniciarse para mantener la aplicación en funcionamiento y que procesos deben lanzarse que, por una razón u otra, no podían lanzarse durante la creación de la imagen (no tiene sentido enlazar volúmenes o arrancar apache en la construcción, ya que una vez lanzados puede que el entorno haya cambiado).
Para la creación de estas imágenes se construyó un script de ruby que automatizaba el proceso al cuál llamamos "ez", pero durante el desarrollo fue evolucionando en lo que hoy sería nuestra herramienta de integración continua. De todas maneras describiremos el proceso de desarrollo para la generación de ficheros que construyen nuestras imágenes y como en la versión final se utilizan para que nuestro script suba páginas en modo producción a la nube.

4.1.1 Apache + aplicación

Empecemos por la imagen que alojará nuestra aplicación y que la expondrá al exterior.
Para la creación de esta imagen decidimos utilizar un servidor Apache y el framework de Magento para alojar nuestra tienda online.
Magento es una plataforma que se encarga de la creación y gestión de una tienda online, liberando al programador de la mayoría de la lógica necesaria para la gestión de productos (base de datos, formularios, usuarios, correos, etc. Todo está gestionado por Magento, a no ser que lo queramos personalizar) y dejando la posibilidad de personalizar el estilo de la página, al mismo tiempo que te permite añadir la lógica que necesites. Otra ventaja de Magento es la gran facilidad que tenemos para añadir sistemas secundarios sin necesidad de programarlos nosotros. Con sistemas secundarios nos referimos a los otros elementos que compondrán nuestro cluster, como son la base de datos, cache, sesiones, etc. Lo único que tenemos que hacer es editar el fichero env.php para configurar la conexión a estos elementos y el framework de Magento se encargará de gestionar la lógica de la tienda para que funcionen junto a ellos.

Apache es un viejo conocido de los sistemas que buscan exponer páginas al exterior. La clave para el desarrollo de nuestra aplicación con este servidor Apache será configurar las extensiones y variables con las que se pone en producción (*memory_limit* es un ejemplo de variable crítica). También será necesario pensar que ficheros de configuración de Apache utilizaremos para desplegar el framework de Magento, los cuales alojaremos en la carpeta */etc/apache2/sites-enabled*, ya que se cargan por defecto cuando se arranca nuestro servidor Apache. En cuanto a que configuración utilizaremos, existen decenas de ejemplos en la web que usan de ficheros .htaccess, pero en nuestro caso simplemente evitaremos que

la configuración de Apache pueda ser sobrescrita con la política *AllowOverride None* y utilizaremos un fichero conf con la configuración necesaria para nuestra página.
Para empezar la etapa de construcción tenemos que pensar en que imagen vamos a utilizar como base para crear nuestro proyecto (FROM). Podríamos empezar con una imagen de Magento, pero uno de los problemas que vamos a encontrar es que no existen imágenes oficiales de Magento, por lo que tendríamos que partir de una imagen creada por otro usuario para iniciar nuestra página y aunque esa opción podrá utilizarse para futuras imágenes, no es muy buena idea que nuestra primera imagen del sistema tenga un procedencia desconocida (especialmente para el pilar de nuestro sistema), por lo que enfocamos la imagen por el lado contrario y buscamos una imagen oficial que venga con Apache y PHP (php:7.0-apache). Esta imagen viene con Apache y php configurados y listos para funcionar cuando arranquemos la imagen; sin embargo, tenemos que encontrar una manera de añadir la configuración personalizada a los elementos existentes, al mismo tiempo que instalamos los componentes necesarios para arrancar Magento. Empezando por PHP, queremos instalar una lista de extensiones que vienen en la documentación oficial de Magento, que ayudarán a mejorar el rendimiento y funcionalidad de nuestra página (algunas obligatorias); para eso utilizamos los comandos de la imagen de Docker que hemos descargado docker-php-ext-install y docker-php-ext-configure, seguidos de las extensiones necesarias que queramos instalar. Algunas de las extensiones requieren configuración adicional, que se puede añadir en el fichero php.ini, por lo que tenemos 2 opciones: Sustituimos el fichero por completo con la configuración que queremos utilizar (COPY) o usamos herramientas como sed para sustituir líneas del fichero que ya existe y añadimos la configuración deseada (RUN sed … /usr/local/etc/php/php.ini). Para el resto de elementos hay que usar *RUN apt-get update && apt-get install* –y … (el flag –y es necesario ya que hay que recordar que el proceso docker build no es interactivo); con él instalaremos librerías y herramientas, que en nuestro caso incluirán ruby, python, fuse, git, tar, composer y muchas otras.

El siguiente paso sería instalar Magento. La idea inicial era utilizar el comando COPY para copiar la estructura completa del proyecto y luego utilizar el comando chmod para ajustar los permisos de los ficheros que se exponían con Apache (seguido de otros pasos que explicaremos a continuación); y aunque en un principio dio resultado, encontramos un serio problema a la hora de construir la imagen: el tiempo. El proceso de construir la imagen sin el uso de cache podía llegar a superar la media hora y en un sistema de integración continua, este tiempo no es aceptable. Lo primero que notas cuando inicias el proceso de construcción es el tiempo que se tarda en construir el contexto de la imagen; Docker genera un contexto con todos los ficheros y carpetas que se encuentran a la altura de nuestro Dockerfile, llegando a generar gigas de contenido que luego no se va a utilizar, por lo que el primer paso fue crear un fichero .dockerignore que excluya aquellos elementos que no son necesarios para la creación de nuestra imagen. Esta exclusión fue muy efectiva, ya que redujo drásticamente el tiempo de construcción; pero aún así pudimos notar otra característica importante a la hora de construir la imagen, y esta sería la cache. La manera en la que la cache de Docker funciona es que cuando la ejecución de uno de los pasos del Dockerfile no cambia, no se ejecuta el comando, sino que se parte de una imagen intermedia que se creó la primera vez que se ejecutó el comando. El problema de este sistema es que cuando un paso cambia, toda la cache de los pasos posteriores (independientemente de que fuera necesario volver a ejecutarlos o no) se pierde y se vuelven a ejecutar; esto ocasiona problemas, ya que el cambio en un solo fichero en uno de los primeros pasos puede provocar que toda la imagen se tenga que reconstruir. Para mitigar este fallo, debemos mover los comandos que no son propensos a cambiar al

principio de la imagen; mientras que aquellos que suelen cambiar debido a la funcionalidad de integración continua (COPY) los movemos al final del fichero, de tal manera que ya no es necesario volver a ejecutarlos.

(*Referencia C1)

Para explotar todavía más el uso de cache, dividimos la imagen en múltiples imágenes dependientes, es decir, la imagen base parte de la de Apache e instala todos los componentes, mientras que las posteriores en vez de partir de Apache parten de la básica y solo cambian si la base lo hace (ahorrando todo el proceso de instalación de componentes).

Para finalizar estos ajustes hemos tomado la alternativa de en vez de copiar la carpeta de Magento entera a la imagen, instalamos con composer un proyecto base de Magento y luego copiamos solo los ficheros esenciales de nuestras página, ya que los comandos COPY requieren mover los ficheros de nuestra máquina a la imagen y son mucho más lentos que los comandos RUN que solamente trabajan dentro de nuestra imagen.

(*Referencia C2)

Continuando con la fase de construcción, tenemos que hablar de la generación de estáticos y la puesta en producción de un proyecto de Magento. Actualmente, para la generación de contenido estático en las páginas utilizamos gulp, que utilizando las urls de la base de datos genera los ficheros necesarios; el problema viene a la hora de conectar a la base de datos, ya que asumimos que no tenemos acceso a la base de datos del cluster (ni querríamos ya que alterar las urls de las bases de datos en producción podría ser peligroso), por lo que para realizar el proceso de producción necesitaremos cambiar el fichero env.php, para que apunte a nuestra base de datos en local (a nuestra IP ya que poner localhost implicaría apuntar al contenedor de nuestra imagen) y tras realizar el proceso que apuntase a la base de datos en producción. Para conseguir esto, generamos una plantilla para el fichero env.php y creamos 2 ficheros, uno para producción y otro para la construcción de la imagen y los cambiamos al final del procedimiento. Utilizando el principio anterior vamos a duplicar también algunos de los ficheros xml básicos de una estructura de Magento y composer.json; ya que con estos ficheros se pueden producir cambios importantes, que en caso de una modificación por error o configuración inadecuada, podrían dejar nuestra página fuera de funcionamiento durante unos cuantos minutos. Cabe añadir que con el uso del comando ARG generamos variables, que podemos utilizar como variables de entorno, que desaparecerán una vez subidas a producción y que podemos setear por consola cada vez que vayamos a crear la imagen, lo que nos permite definir la versión de las aplicaciones que vamos a utilizar en cada momento, sin necesidad de alterar el código del Dockerfile.

Tras esto llega la fase de producción/ejecución. A la hora de crear la imagen tenemos que tener en cuenta que nuestro entrypoint se ejecutará cada vez que una imagen se arranque; así que aprovecharemos para editar el script y que monte todos los volúmenes de nuestra imagen, al mismo tiempo que damos la posibilidad de ejecutar otro tipo de comandos con la línea exec “$@”. El motivo por el que usamos volúmenes dentro de nuestras aplicaciones web es para mantener aquellos ficheros que tienen que ser idénticos entre todas las instancias de nuestro contenedor; un ejemplo de esto sería la carpeta pub/media de Magento, que contiene todas las imágenes de nuestros productos; si una cambiase queremos que todos los contenedores reflejen ese cambio, por lo que montamos un

volumen en la carpeta y nos aseguramos de que todos los ficheros de imágenes sean iguales en las distintas instancias de nuestra web (más información en la sección de S3).

Para finalizar, además de la duplicación de ficheros para evitar errores al subir nuestra página a producción, hay un problema de seguridad que debemos atacar y en este caso serían los logs de la aplicación. Por defecto, una vez que subimos nuestra imagen al cluster, la salida de la terminal puede ser vista con comandos kubectl, por lo que la imagen de Apache que utilizamos crea enlaces simbólicos de los logs hacia /dev/stdout, pero; ¿qué pasa con los logs no pertenecientes Apache? Una manera de atacar la situación sería hacer lo mismo y redirigir las salidas, pero queríamos llevarlo un paso más allá; queríamos recibir alertas cada vez que se produjese un error en la página, por lo que utilizamos la tecnología de cloudwatch para monitorizar los logs y poder verlos desde la consola de Amazon; pero esta alternativa resultó ser demasiado cara, ya que se nos cobraba por cada contenedor la cantidad de información que se pasaba a Cloudwatch. Contando que cada contenedor puede estar duplicado varias veces y que existen varias páginas, la factura final resultaba inviable para nuestro proyecto. Como alternativa, decidimos utilizar la tecnología de Sentry, que permite monitorizar solo los errores de PHP y enviarnos alertas al correo cada vez que estos ocurran; además existía una extensión del programa compatible con Magento, por lo que solo tuvimos que añadirla a la carpeta extensions de nuestros proyectos y configurarla en la base de datos en producción; de esta manera, teníamos 2 métodos para debugear nuestra aplicación en caso de error o ataque, lo cual aumentó considerablemente la seguridad de nuestra aplicación.

4.1.2 Redis

Redis es una aplicación que almacena estructuras de datos, que utilizaremos para guardar las sesiones de nuestras páginas y como sistema de cache. Gracias a la compatibilidad de Magento con Redis podemos modificar el fichero env.php, para que almacene los ficheros de sesión en este nuevo contenedor, en vez de en local; ya que si seguimos esa opción, podemos encontrarnos con que nuestros clientes pierden su sesión, cuando al conectarse a nuestra página son redirigidos a contenedores distintos.

La imagen de esta aplicación es muy sencilla, ya que al partir de la imagen oficial de Redis en Dockerhub tenemos todo lo necesario para ponerlo en funcionamiento ya instalado. Lo único que tenemos que hacer para adaptarlo a nuestras necesidades es sobrescribir el fichero */usr/local/etc/redis/redis.conf* y cambiar el CMD para que utilice ese fichero como configuración de arranque. Nada más sería necesario para tener nuestra imagen de Redis creada; pero hay algunos parámetros que modificaremos para aumentar la seguridad de nuestra aplicación y también para hacerla compatible con la nueva arquitectura. Para empezar, cambiaremos el campo *maxmemory* a 1gb, ya que al ser un programa pesado en uso de RAM queremos que solo sea capaz de crecer hasta cierto punto, antes de que el algoritmo LRU empiece a borrar ficheros; también activaremos el *protected-mode*, para que sea obligatorio introducir una contraseña al conectarse por remoto, la cual tendrá que ser idéntica a la del campo requirepass. Por último, modificamos el campo *bind* para que se escuchen las comunicaciones de cierto rango de IPs; lo pondremos a 0.0.0.0 para que escuche todas las direcciones (hay que recordar que sigue siendo necesario poner una contraseña para acceder y que si no exponemos el contenedor al exterior solo los elementos internos al cluster podrían acceder, por lo que no perderíamos en seguridad), aunque podría ser asignado al conjunto de IPs internas del cluster, que se hayan definido en el security group de Amazon o a los pods donde se alojarán nuestros contenedores de redis; el problema con esta última implementación viene a la hora de obtener ese rango de IPs a

asignar y que nos veríamos obligados a reconstruir la imagen en caso de que el rango cambiase, dejándonos sin sesiones en las páginas web hasta que lo solucionásemos (el error sería todavía peor, ya que las sesiones podrían almacenarse en local hasta poder volver a conectarse y obtendríamos fallos intermitentes más difíciles de diagnosticar).

4.1.3 Varnish

Varnish es un sistema de cache para peticiones HTTP, que servirá para aligerar la carga sobre nuestros servidores Apache. La idea es utilizar esta aplicación para cachear todo el contenido estático de nuestra página, de tal manera que cuando se produzcan incrementos en el tráfico de la página o se produzca un ataque de denegación de servicio Varnish, devolverá repuestas cacheadas realmente rápidas, que evitarán que lleguen a nuestro servidor y mitigarán el impacto sobre nuestro sistema, al mismo tiempo que seguiremos siendo capaces de dar servicio a nuestros clientes.
La imagen de Varnish es un poco más compleja que la de Redis, ya que requiere partir de la imagen de *debian* para instalarla, debido a que no existe una imagen oficial para esta aplicación durante el periodo de desarrollo de este TFG.

(*Referencia C3)

La creación de esta imagen sigue un proceso similar a la de Apache, en la que instalaremos la aplicación de varnish con apt-get install y la lanzaremos en nuestro entrypoint con los parámetros de configuración necesarios (también es posible hacerlo con un fichero de configuración); entre ellos asignaremos la memoria a utilizar por nuestra máquina, la ruta a la configuración de la cache del back y el puerto donde escucha peticiones (lo ponemos en el 80, ya que queremos que funcione como intermediario entre nuestro Apache y el cliente sin que note la diferencia).

(*Referencia C4)

Respecto a la configuración para cachear nuestra página, podemos encontrar la estructura base en la red y luego añadir las modificaciones que creamos convenientes sobre el fichero. El fichero es extenso, pero las líneas más relevantes para nuestro sistema son las siguientes:

(*Referencia C5)

El backend representa la página destino que deseamos cachear, mientras que acl purge refleja el conjunto de IPs que pueden limpiar la cache. Originalmente teníamos pensado usar distintos backend para cada página (.host refleja el dominio donde está nuestra página) y filtrarlos por url usando la línea req.url ~ “URL”, pero esta idea pronto quedó descartada, debido a un par de razones: si Varnish caía (aunque gracias a Kubernetes se recuperaría automáticamente) perdíamos acceso a todas las páginas de nuestro cluster (además de ser un cuello de botella del tamaño de nuestro cluster) y cuando queríamos añadir una nueva página, editarlas o borrarlas teníamos que volver a construir la imagen y desplegarla, por lo que no solo perdíamos las páginas del cluster durante unos minutos, sino que toda la cache almacenada de todas las páginas se perdería con cada cambio o caída. Al final optamos por usar namespaces separados, de tal manera que podíamos usar el mismo dominio para todas las páginas, sin tener que cambiar la imagen de varnish y aislando los distintos entornos de cada página.

Como medida adicional, para incrementar la seguridad de nuestra página podemos añadir un filtro de peticiones en nuestro Varnish que analice las urls y busque intentos de inyección SQL que están automatizados por bots y evite el acceso a partes más críticas de nuestra web (como son la base de datos o el servidor). Un ejemplo de líneas que podríamos añadir para filtrar algunas peticiones son:

(*Referencia C6)

Con estas líneas evitamos que pasen algunos de los parámetros más frecuentes en las peticiones maliciosas (select, unión, sleep,…), pero hay que tener en cuenta que esta no es la manera completa de defenderse ante un ataque; el uso de sentencias preparadas y la validación de entradas siguen siendo críticas para evitar la inyección y, si nos excedemos con las filtraciones de parámetros, podemos provocar que peticiones válidas ya no sean capaces de llegar a nuestra página web. Aunque el uso de estas filtraciones reducirá carga de trabajo innecesaria en nuestros sistemas.

Para configurar Magento para que apunte a Varnish, utilizamos el mismo proceso que con Redis; modificamos el env.php para que apunte al dominio de Varnish.

Por último cabe destacar que tener un sistema de cache antes de llegar a nuestro servidor añade una capa extra de seguridad a nuestro sistema ya que evitamos el acceso directo al servidor (que con el uso de Kubernetes aislamos incluso el contenedor del exterior) y respondemos a peticiones con respuestas rápidas cacheadas que evitan una sobrecarga en el contenedor de Apache. Por poner un ejemplo, fuimos capaces de mitigar un exceso de peticiones a una de las páginas de nuestro cluster gracias a este componente debido a su respuesta de elementos cacheados a las peticiones entrantes; evitando que llamasen al pod de Apache y (en el caso de Kubernetes) evitando la duplicación innecesaria de recursos para atender a un gran incremento de peticiones.

4.1.4 Mysql

Mysql funcionará como la base de datos de nuestra página. Como con Redis, podemos partir de la imagen oficial de mysql:5.7 y modificar un solo fichero de configuración con el comando copy, que se cargará al iniciar la imagen (*/etc/mysql/conf.d*). Uno de los parámetros que debemos tener especialmente en cuenta es max_connections, debido a una característica especial de nuestra arquitectura; normalmente simplemente ajustaríamos el número máximo de conexiones acorde al tráfico de nuestra página y la capacidad de nuestro servidor, pero ahora hay que tener en cuenta que los contenedores serán capaces de autoreplicarse y podemos llegar a tener múltiples instancias de la página, cada una con sus peticiones a la base de datos, lo que puede ocasionar que nos quedemos sin servicio debido a una saturación de peticiones a la base de datos.

Aunque con esa configuración debería ser suficiente, también hemos querido implementar una modificación, para que en un futuro sea posible tener la base de datos con instancias maestro-esclavo, en las cuales solo una sea de escritura (maestro), mientras que el resto sirvan para operaciones de lectura, haciendo posible la replicación de la base de datos sin que afecte a su consistencia e integridad (Magento también permite el uso de esta configuración en el env.php, pero de momento solo está disponible en la versión ecommerce).
Para implementar esta funcionalidad, vamos a querer cambiar el entrypoint de nuestra imagen; podemos hacerlo editando directamente el fichero que se ejecuta con el uso del

comando sed o sustituyéndolo por completo con un COPY (algunos entrypoint permiten insertar comandos personalizados, incluido este, pero el momento en el que se ejecutaban no era el deseado). Optamos por la segunda vía, pero como no queríamos perder la funcionalidad ya implementada, buscamos dentro del Github el entrypoint que se ejecutaba previamente y le añadimos algunas modificaciones. Por defecto. el entrypoint solo construye la base de datos desde cero si el directorio de /var/lib/mysql está vacío o la inicia en el caso contrario; por lo que queremos encontrar donde se crean los usuarios y añadimos estas líneas:

(*Referencia C7)

Las líneas en la imagen master de mysql crean el usuario slave y le dan acceso desde el exterior, mientras que en la imagen slave utilizamos ese usuario para conectarnos a la base de datos y sacar una réplica de la misma que, aunque puede ser modificada, no produce cambios en la base de datos real.
Como se puede observar, en el extracto del código estamos utilizando variables de entorno que definiremos en Kubernetes, como es el host de la base de datos del master o el usuario a utilizar (algunas de ellas son creadas por nuestra imagen, como sería el usuario, mientras que las de los host las crea automáticamente Kubernetes, una vez que iniciamos el contenedor en un pod dentro del cluster).

4.2 Servicios en la nube

Una vez tenemos creadas las imágenes que vamos a usar en nuestro cluster, queremos tener un entorno donde poder alojarlas y distribuirlas al público. Gracias a los servicios en la nube podemos hacer justo eso, ya que estos nos proporcionarán los recursos necesarios para crear nuestro sistema y pondrán a nuestra disposición herramientas que podemos utilizar para adaptar la funcionalidad de las páginas anteriores. Para el desarrollo de nuestro entorno hemos decidido utilizar Amazon Web Services y vamos a describir algunos de los servicios que proporciona, los cuales sirven a propósitos críticos de nuestro sistema, que utilizaremos una vez que lleguemos a los sistemas distribuidos.

4.2.1 Repositorios

Amazon dispone de un servicio de contenedores en línea, que nos da la posibilidad de guardar nuestras imágenes de Docker en repositorios, conocidos como los ECR.
Su finalidad es simple, una vez que nuestra herramienta termine de crear la imagen de la aplicación, crearemos un repositorio desde la consola de Amazon, que nos devolverá un URI al que podemos apuntar cuando vayamos a subir la imagen (utilizaremos el comando docker push para subirlas al repositorio).

La idea inicial es tener un repositorio para Varnish, otro para Redis, otro para MYSQL y, para finalizar, repositorios independientes para cada una de las páginas donde pondremos nuestros servidores Apache. El motivo de necesitar un solo repositorio para algunos componentes es que podemos reutilizar las mismas imágenes para distintas páginas sin tener que modificar ninguno de sus parámetros, por ejemplo, Varnish apunta a un dominio llamado project-service, que puede ser asignado a cualquier tipo de página web que están separadas por namespaces (por lo que pueden tener el mismo nombre, ya que las variables de entorno que utiliza Kubernetes para resolver dominios serían distintas).
La utilidad de los repositorios no acaba aquí ya que incluyen una funcionalidad que será crítica para la seguridad de nuestra página. Cada vez que subimos una imagen se le añade

un campo digest con un identificador único (estilo: @sha256:<randomstring>), lo cual nos permite implementar una funcionalidad de rollback en subidas de nuestras imágenes con mucha facilidad. Por poner un ejemplo, supongamos que se sube una imagen con un error crítico que provoca que se cierre el servidor Apache de manera constante, en vez de tener la página caída mientras encontramos la causa del error, podemos actualizar la imagen desplegada a una versión previa, usando el digest que se generó al subir la imagen y cuando encontremos el error actualizar la nueva imagen, sin que se note una caída significativa del servicio.

Un factor extra que hay que considerar es que cuando subimos una imagen, por defecto, viene con una etiqueta conocida como latest, que se sobrescribe cada vez que subimos la imagen al repositorio. En un principio no debería haber ningún problema con esta implementación, pero debido a la integración continua donde nuestra página está en constante funcionamiento, hay que tener en cuenta que cuando una imagen se cambia sobre el mismo tag, las imágenes ya arrancadas no son conscientes del cambio en el repositorio, por lo que es necesario volver a pullear la imagen manualmente, cada vez que realicemos cambios (también provoca que no sepamos si de verdad está la imagen actualizada a primera vista). Para solucionar este problema hacemos uso de los tags; etiquetas personalizadas que guardaremos en un fichero y que serán más fáciles de identificar que los digest, que además se aseguraran de que la imagen esté actualizada, ya que cuando se lanza una imagen con un nuevo tag, la antigua es sustituida (todo el proceso automatizado en nuestro script). Además, con el uso de tags podemos crear 2 tipos de imágenes para la base de datos, usando un mismo repositorio en el que un tag se usa para los maestros y otro para los esclavos.

4.2.2 Cluster

El cluster es el entorno que contendrá todos los recursos virtuales que alojarán nuestros contenedores. Para crear un cluster necesitaremos una VPC, que nos permitirá lanzar nuestras instancias EC2 en una red virtual privada, aislada de elementos externos, con su propio conjunto de IPs que se utilizarán para comunicar las instancias del interior del entorno (cada una con su grupo de seguridad, que actúa como un firewall que controla el tráfico de salida y entrada a las instancias que lo componen). Este elemento se puede crear desde una consola de AWS, simplemente especificando su región y el conjunto de subredes que la formarán (Irlanda, por ejemplo, está dividida en las zonas eu-west-1 a, b y c).
Con la VPC creada, pasamos a la creación del Amazon EKS Cluster. Al igual que el elemento anterior, creamos el cluster desde la consola de Amazon pasándole la VPC, security group y subredes que hemos creado previamente (según las que elijamos, estableceremos las zonas en el que las instancias se crearán); también será necesario especificar un rol IAM, que contendrá los permisos necesarios para operar un cluster EKS (los roles son conjuntos de permisos que podemos asignar a usuarios y entidades de AWS, que en nuestro caso necesitaremos uno que tenga los permisos de la sección EKS, que permiten crear recursos en nuestro lugar). Con estos elementos ya tendríamos un cluster con la capacidad de gestionar nuestro entorno Kubernetes por nosotros; EKS es esencial para el objetivo de seguridad y resiliencia que queremos alcanzar, ya que es capaz de crear recursos en distintas zonas, para evitar la caída del cluster en caso de un error de servidores en una localización (garantiza alta disponibilidad), de sustituir instancias dañadas por nosotros (aunque se presentan casos en los que si kubelet cae, podemos quedarnos sin respuesta de estado de una instancia) y de mantenerlas actualizadas, de crear balanceadores

de carga que nos ayudarán a controlar el tráfico entrante y de la gestión de permisos y redes aisladas, con la mínima intervención por parte del administrador de sistemas.

Con EKS listo nos faltaría asignar las instancias EC2 o máquinas virtuales que formarán nuestro cluster y que nos permitirán alojar nuestros contenedores, también conocidas como worker nodes. Como antes, abriremos la consola de AWS en cloudformation y seleccionaremos el template de la URL https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2019-02-11/amazon-eks-nodegroup.yaml, que nos permitirá rellenar los campos necesarios para el grupo de trabajo que necesitamos (los campos requeridos serán el cluster donde queremos alojarlo, el security group, VPC, subredes, etc.); pero los campos que nos interesan serán el número mínimo y máximo de nodos, junto con el tipo de instancias que serán generadas; ya que una vez que lleguemos al autoescalado podremos observar como Kubernetes es capaz de crear y borrar instancias según el uso de los recursos que se tiene en cada momento, ayudándonos a reducir los costes de nuestro entrono. Una vez creado el grupo, nos apuntamos el NodeInstanceRole y nos descargamos el template de la URL https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2019-02-11/aws-auth-cm.yaml y con él cambiamos el rolearn por el de nuestro grupo, lo cual permitirá que las instancias creadas por este grupo de autoescalado se unan a nuestro cluster y sean gestionadas por el sistema Kubernetes.
Para añadir esta configuración, necesitamos que kubectl apunte al cluster creado, por lo que usaremos el comando aws update-kubeconfig --name <cluster> (puede que necesites añadir --region) para actualizar el entorno en nuestro cliente y haremos un kubectl apply –f para aplicar el yaml anterior; con lo anterior tendríamos todo lo necesario para crear nuestro sistema distribuido.

Como nota adicional, es posible editar el yaml que pasamos a la consola de AWS. Un parámetro a cambiar podría ser el tipo de instancia que utilizamos, pudiendo utilizar lo que se conocen como spot-instances, que son nodos de subasta ofrecidos por Amazon, que son mucho más baratos, a cambio de que puedan ser requisados en cualquier momento por Amazon, en caso de que los necesiten. Para realizar el cambio, podemos partir del yaml de demanda (https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2018-11-07/amazon-eks-nodegroup.yaml) y añadir los campos de las instancias spot, entre los que se incluyen el spotprice, el type de instancia con el valor de Spot a true y los argumentos al script de bootstrap, que pueden incluir los labels que utilizaremos para identificar los nodos (tipo --node-labels=cpuspot). Si no queremos partir a ciegas, podemos descargarnos ejemplos de un cluster que utiliza ambos tipos y coger los elementos relacionados al grupo de spot instances (https://github.com/awslabs/ec2-spot-labs/blob/master/ec2-spot-eks-solution/provision-worker-nodes/amazon-eks-nodegroup-with-spot.yaml); aunque nada de esto es realmente esencial para tener nuestro cluster funcionando.

Para añadir una capa de seguridad extra a nuestro cluster podemos modificar el yaml de autorización dentro de nuestro cluster, el cual podemos indentificar con el comando *kubectl describe configmap -n kube-system aws-auth*. Este fichero tiene 2 secciones: roles y usuarios. El primero se utiliza para elementos del cluster, como los grupos de nodos y roles de usuarios; y el segundo para permisos específicos de los usuarios. Ambos tienen un identificador (en AWS es el arn del elemento) seguido de un nombre y por último un grupo de permisos en el cluster que definen las acciones que puede realizar un usuario en el cluster; asociados con un objeto tipo ClusterRoleBinding que apunta a un objeto ClusterRole que define los recursos y tipo de permisos que tenemos sobre ellos (todo en yamls).

Esto no acaba aquí ya que gracias a los roles IAM de AWS podemos definir políticas sobre objetos del proveedor de servicios en la nube usando su arn y una lista de acciones permitidas, controlando así quién tiene acceso según su rol en el desarrollo de la app.
(*Referencia C17)

4.2.3 S3

Otra de las características a las que tenemos acceso en AWS es el almacenamiento seguro en la nube que nos proporciona S3. Como ya hemos comentado previamente, en la imagen de Apache necesitamos que ciertos ficheros se puedan compartir entre contenedores duplicados, ya que si no fuera así tendríamos problemas de consistencia, en los que algunos clientes podrían ver las imágenes, mientras que otros obtendrían un error; además S3 dispone de metadatos y ajustes de visibilidad, que nos permiten indicar que ficheros son públicos y se pueden acceder por URL (útil en Cloudfront) y cuáles no. Gracias a s3fs-fuse podemos montar en el directorio donde almacenaremos el contenido multimedia, un bucket de S3, como si de un sistema de ficheros se tratase; por lo que durante la ejecución del entrypoint de nuestra imagen (no puede hacerse en la etapa del build) utilizaremos el siguiente comando:

(*Referencia C8)

En el parámetro bucket y path ponemos el nombre de nuestro bucket, junto a la ruta interna dentro del mismo, hasta nuestra carpeta con los ficheros, seguido de la ruta de nuestra imagen a la carpeta donde Magento guarda el contenido multimedia. Por último, especificamos las opciones que pueden incluir, entre otras, use_cache para especificar en qué directorio se guardará, max_state_cache_size para el número de entradas en caché máxima, junto con stat_cache_expire para especificar el tiempo en segundos que tarda en expirar la cache y muchas otras.
Como funcionalidad añadida, también lo utilizaremos para almacenar backups de nuestras bases de datos de manera periódica, utilizando cronjobs (se explicará en sistemas distribuidos).

4.2.4 Cloudfront

Cloudfront es un servicio CDN o red de distribución de contenidos, que nos permite copiar todo nuestro contenido estático a distintos sistemas, que se pueden encontrar fuera de nuestro cluster y que llegan al usuario final como si se tratase de una petición normal al servidor; además, es compatible con otros servicios de Amazon, como S3, lo que hace su configuración bastante más simple. El objetivo de este servicio es aliviar la carga sobre nuestros contenedores, haciendo que otros sistemas pasen contenido estático, al mismo tiempo que aumentamos la disponibilidad y la velocidad de respuestas de nuestras páginas, al llevar el contenido a centros de distribución que pueden estar más cerca del cliente y que en caso de caída pueden ser sustituidos por otros.

La configuración que utilizaremos para nuestros CDN será la siguiente: 2 orígenes, uno apuntando a toda la carpeta pub de Magento, la cual aceptará conexiones de tipo HTTP o HTTPS (según si el cliente quiera o no conexión segura) y otro apuntando directamente a un origen de tipo s3, que será aquel que contiene los ficheros que hemos comentado en apartados anteriores; y como comportamiento para las rutas pondremos que la dirección de la página que apunta a /static/ vaya al dominio de nuestra página a web (Varnish y Apache), mientras que la que apunta a /media/ en vez de ir a nuestra página vaya

directamente al bucket de S3. Ambas comprimirán sus objetos automáticamente y añadiremos a las cabeceras de los paquetes el campo Origin que nos permitirá CORS, ya que los elementos de nuestra página vendrán de distintas fuentes.

4.2.5 Volúmenes EBS

Para finalizar, tenemos que hablar del almacenamiento en disco de nuestras instancias EC2 que forman nuestros nodos de trabajo. Uno de los campos que tuvimos que rellenar a la hora de crear los grupos de trabajo que se unirán al cluster fue el tamaño de los volúmenes que se unen a nuestros nodos; esto es debido a que, aunque no nos diésemos cuenta, cada vez que se crea un nodo se crea un volumen de almacenamiento EBS, que se unirá a nuestra máquina virtual y funcionará como sistema de ficheros. Este concepto nos interesa ya que los volúmenes EBS ofrecen la posibilidad de crear un almacenamiento permanente (los creados automáticamente con los nodos desaparecen al borrarlos, debido a la configuración por defecto del yaml que aplicamos en apartados anteriores), que está pensado para ofrecer baja latencia y obtener altos niveles de rendimiento, lo cual es muy útil para la creación de bases de datos como la que utilizaremos en nuestra imagen de MYSQL.

Su creación en la nube no tiene ninguna dificultad, ya que desde la consola de AWS podemos crear uno manualmente, simplemente definiendo el tipo de almacenamiento y su tamaño (se pueden crear automáticamente desde Kubernetes con el uso de PersistentVolumes y los Claims correspondientes), pero hay que tener en cuenta que debido a la implementación de la imagen de mysql de Docker, tenemos que estar seguros de que el volumen está vacío la primera vez que creamos nuestra base de datos, lo cual es sencillo ya que podemos conectarnos por ssh a la instancia y borrar el contenido que haya podido ser creado por defecto o borrarlo desde la imagen con un comando en el entrypoint o montando el volumen desde cualquier otro elemento (teniendo en cuenta que solo puede haber un sistema conectado al dispositivo a la vez y que en el caso de las instancias de EC2 de momento solo son compatibles si se encuentran en la misma zona). De esta manera nos aseguramos de que nuestra base de datos no se pierda una vez que el pod que contiene MYSQL se cierre por el motivo que sea.

4.3 Sistemas distribuidos

Finalmente entramos en la herramienta clave que se encargará de gestionar las partes más complejas de los sistemas distribuidos, ya que gracias a la herramienta Kubernetes la replicación de recursos, el balanceo de la carga, el descubrimiento de las IPs y la conexión de los elementos del cluster quedan abstraídos al programador y dejan en nuestra mano la definición de los recursos que queremos gestionar dentro de nuestro cluster. Estos recursos se pueden generar usando comandos kubectl o, como alternativa menos engorrosa, usando ficheros yaml, que siguiendo la estructura definida por el comando *kubectl explain* consiguen el mismo resultado.

4.3.1 Pods y deployments

Como hemos comentado en otras secciones la unidad básica de Kubernetes es el pod el que aloja los contenedores que contendrán las aplicaciones necesarias para el funcionamiento de nuestra web. Sin embargo, vamos a ir un paso más allá y vamos a empezar con los deployments, ya que implementan una de las funcionalidades que más vamos a utilizar en nuestro cluster: la replicación.

(*Referencia C9)

Los deployments son conjuntos de pods idénticos, que tienen la capacidad de funcionar en conjunto con los pods que pertenecen al mismo grupo y que gracias a los labels podemos identificarlos como elementos del mismo grupo y redirigir todo el tráfico a ellos. Gracias al parámetro replicas podemos definir el número de pods que queremos que se ejecuten, el cual podemos cambiar según el número de peticiones que lleguen a nuestro sistema y así ser capaces de ajustarnos a la demanda del servicio sin perder velocidad en la respuesta al cliente.
El parámetro spec define las propiedades que tendrá nuestro conjunto (en este caso el número de replicas y el selector que utilizaremos para identificar los pods que pertenecen a nuestro grupo) y los spec anidados definen las propiedades de los elementos que componen nuestro recurso (en el caso de los deployments serían los pods y en el caso de los cronjobs serían los jobs). Respecto a las propiedades de los pods, podemos destacar el nodeSelector que, de la misma manera que los labels, nos permite seleccionar aquellos nodos que tengan la etiqueta que ha sido especificada y solo asignar nuestros pods si se cumple la condición especificada (esto funciona especialmente bien con la parte que hemos mencionado previamente de spot-instances, en el que podemos asignar pods con funciones que no sean críticas a un tipo de nodo spot-instance más inestable mientras que otras partes se asignan a nodos de demanda que raramente caerán), de tal manera que podemos tener nodos con distintas características (más CPU o memoria) y asignar nuestros pods según lo que necesiten. El otro parámetro que podemos definir son los contenedores, que puede tener nuestro pod, de tal manera que podemos tener varias imágenes de Docker en un solo pod sin tener que usar uno por recurso; pero a pesar de esto hemos decidido separar las imágenes ya que si un pod se cae solo perderíamos una aplicación en vez de todos los recursos de la página. En los parámetros de los contenedores podemos definir la imagen y su tag (recordemos que el tag es útil ya que nos sirve para volver a una imagen previa en caso de error muy fácilmente y en poco tiempo), los cuales alteraremos con una aplicación en Ruby cada vez que queramos subir cambios en nuestra página. Existen también parámetros, como el puerto, donde se expone la aplicación del contenedor o el securityContext que nos permite realizar operaciones extra como el uso de volúmenes internos en la aplicación (los bucket de S3 en modo volumen de ficheros que hemos comentado en apartados anteriores).
Un campo crítico que podemos añadir a los contenedores son los recursos que va a utilizar. Los dos recursos básicos que podemos encontrar en los nodos son los de CPU y memoria, que podemos asignar como requests o limits. Los requests son los recursos que vamos a reservar para nuestro contenedor y que podemos estar seguros de que siempre estarán disponibles cuando los solicitemos, mientras que los limit definen un tope a los recursos que podemos usar una vez que el contenedor necesite más recursos de los que tiene asignados de tal manera que no crezca descontroladamente en caso de que haya un exceso de tráfico. Los limits se comportan de manera distinta, según el tipo de recurso que se utiliza; en el caso de la CPU nuestro contenedor queda limitado al limit que se haya especificado, mientras que en el caso de la memoria kubelet identifica los pods que han superado la RAM definida en request y los pone como candidatos para ser expulsados en caso de que no haya memoria suficiente para todos los pods del nodo; y en el caso de superar el limit son rápidamente expulsados del sistema y vuelven a usar la memoria RAM que tenían definida en su request. Aquellos pods que no tienen definidos sus recursos son considerados best-effort y son los primeros en ser expulsados por kubelet en caso de necesitar más recursos, pero a cambio son capaces de usar los recursos disponibles en el nodo; después de estos están los burstable que son aquellos que tienen los requests y limits

definidos, pero no son iguales de tal manera que pueden usar más recursos cuando los necesitan y siempre tienen un mínimo de recursos definidos en caso de que hay un exceso de uso en el nodo (tienen un comportamiento más impredecible). Por último estarían aquellos que tienen sus recursos definidos y son idénticos en sus requests y sus limits, de tal manera que siempre tienen cierta cantidad de recursos y como máximo pueden utilizar aquellos que les han sido asignado; éstos son conocidos como guaranteed (como nota adicional, si no definimos los limit por defecto se ajustan a la capacidad máxima del nodo).
Dejando los recursos de lado, pasamos a las variables de entorno, como recordarán las variables de entorno pueden ser definidas dentro de los Dockerfile con el comando ENV y pueden ser utilizadas dentro de nuestro entorno para definir direcciones o elementos similares (incluso Kubernetes hace uso de ellas para definir las IPs de los recursos que se encuentran dentro del mismo namespace de nuestro pod), simplemente pasándoles el valor que queremos asignarles. Estas variables de entorno pueden utilizarse en conjunción con otro componente, llamado Secret, que nos permite asignar información sensible a las variables de entorno, sin que pueda ser vista cuando describimos los recursos del cluster.

(*Referencia C10)

Existen campos especiales, como los volumemounts, que nos permiten declarar el uso de volúmenes externos que se montarán en un directorio específico dentro de nuestro pod. Para el correcto funcionamiento de estos volúmenes tenemos que declarar un campo llamado volumes, con el nombre del volumen definido en la sección de volumemounts y asociarlo con un recurso llamado PersistentVolumeClaim, que es una solicitud de volumen que cumpla ciertas características especificadas, como son el tipo y tamaño de almacenamiento. Estos volúmenes pueden ser definidos con el recurso PersistentVolume, que están pensados para responder a las peticiones definidas previamente y asociarse a ellos creando un volumen que cumpla esas características.
Normalmente habríamos utilizado esos componentes para nuestras bases de datos, pero preferimos crear el volumen a mano y en el apartado de volumes poner su ID (el motivo por el que lo hicimos así es debido a que la creación automática de volúmenes puede generar contenido previo en ese mismo volumen, que provoca que la creación de la imagen de mysql falle al arrancar, al decir que ya existe contenido dentro del directorio donde queremos montarlo):

(*Referencia C11)

Por último estaría el parámetro args que nos permite pasarle parámetros al entrypoint de nuestra imagen, pero entraremos en detalle en la siguiente sección.

4.3.2 Jobs, cronjobs y sus usos

Cuando arrancamos un pod, necesitamos que el contenedor que lo mantiene abierto nunca se cierre, ya que si termina kubelet lo identifica como caído y reinicia la imagen. Esto es especialmente útil cuando un servicio sufre un error y queremos que se recupere sin que tengamos que intervenir, pero puede resultar un problema cuando queramos lanzar un solo comando en nuestra imagen sin dejar el servicio arrancado; y aquí es donde entran los job.
Los job funcionan de manera idéntica a los pod, pero con una diferencia: están pensados para ejecutarse el número de veces que especifiquemos (incluso de manera concurrente) y reiniciarse solo si la salida del entrypoint devuelve un código que no sea 0.
A primera vista, no parece algo que no se pueda conseguir conectándose a un pod de manera remota y lanzando un comando en una terminal, pero ¿qué ocurre con aquellas

tareas periódicas que queremos ejecutar cada cierto tiempo?: es aquí donde entran los cronjob. Por norma general nada nos impide instalar un cronjob en la imagen de nuestro pod y dejar que se ejecute cada x tiempo, pero hay que pensar que si instalamos una tarea en una imagen no se ejecutará una sola vez, sino que cada replica de la imagen lo ejecutaría en el tiempo especificado haciendo que si, por ejemplo, tenemos 4 replicas, el comando se ejecutaría una vez en cada una, llegando a ocasionar problemas graves si no se pueden ejecutar varios a la vez o realizando operaciones innecesarias.
Es aquí donde entra en juego el recurso cronjob, que funciona de manera similar a los cronjob de Linux; generan jobs (los cuales se asignan a pods) cada x tiempo con la plantilla que les pasemos.

(*Referencia C12)

Una forma de utilizar los cronjobs es reutilizar el código de Magento, pero añadiendo una modificación, que es pasar argumentos con args. La idea es pasarle el comando que queremos ejecutar por yaml y en el entrypoint la línea exec "$@" se encargará de coger los parámetros y ejecutarlos como si fuesen un comando normal, en vez de arrancar Apache, lo cual cerrará el pod una vez termine el comando que hayamos especificado.
Esto es especialmente útil a la hora de realizar backups de la base de datos de nuestras páginas: especificamos el Schedule a la hora que queramos hacer la copia, por ejemplo, "0 4 * * *", creamos una imagen con mysql-cli con un entrypoint que haga la copia de la base de datos, lanzando un comando mysql apuntando a la dirección del pod con la imagen de mysql (utilizamos una combinación de variables de entorno con un recurso secret que hemos visto en el apartado anterior) y guarde la copia en un volumen y simplemente lo lanzamos en el cluster.

(*Referencia C13)

Con la combinación de s3fs, mysql-cli y sentry para capturar fallos podemos hacer backups de bases de datos todos los días y recibir notificaciones en caso de que haya un fallo al hacerla.

4.3.3 HorizontalPodAutoscaler

La capacidad de replicar nuestros pods, según la necesidad de nuestro servicio, es indispensable si queremos ser capaces de responder a picos en el tráfico; sin embargo con la configuración actual nos vemos obligados a vigilar el servicio constantemente, fijándonos en los tiempos de respuesta de la página y replicando los pods cuando lo consideremos necesario. Esto no tiene porque ser así y podemos hacer uso de los horizontal pod autoscalers, para que vigilen el uso de CPU de los deployments y repliquen los pods cuando superen un umbral especificado. Lo único que tenemos que hacer es apuntar al deployment que queremos vigilar usando su nombre y luego especificar el número mínimo y máximo de replicas, junto con el umbral de uso de CPU que queremos que provoque la replicación (se pone en porcentaje y se calcula respecto al requests definido en el deployment).

Este componente es realmente útil ya que con la replicación de pods podemos ajustar el uso de recursos según el tráfico de la página y obtener al mismo tiempo la reducción de los costes al reservar menos recursos cuando no son necesarios, y el incremento de recursos cuando se incrementa el tráfico y queremos satisfacer toda la demanda sin que afecte al rendimiento de nuestro sistema.

(*Referencia C14)

***Nota:** Para usar este componente hay que tener metrics instalado, ya sea con yamls de Github o con Helm

4.3.4 Services

Una vez que tenemos desplegados todos los pods que contendrán nuestras aplicaciones, necesitamos una manera de que puedan ser accedidos tanto desde el exterior como el interior del cluster y ser capaces de replicar nuestros pods con la misma aplicación pero distintas IPs, ¿cómo somos capaces de acceder a ellos? La respuesta está en los services; estos componentes funcionan como método de descubrimiento de recursos, que permiten acceder a todos los pods de un subconjunto apuntando al servicio que los monitoriza.

(*Referencia C15)

Los services son componentes de Kubernetes a los que se asigna una IP y se introduce en los pods del cluster a modo de variable de entorno, de tal manera que cada vez que se apunta al dominio del nombre del service somos capaces de sacar una IP que redirige a los pods que controla; por lo que, si queremos conectarnos al pod de una página web, no apuntamos a la IP de la máquina que lo contiene, sino que apuntamos a la IP del servicio para que luego el servicio identifique a los pods que tienen el label especificado en el selector, busque el puerto en el que están expuestos (port sería el lugar de entrada al servicio, mientras que el targetPort es el puerto donde está expuesta la aplicación del pod, por lo que podríamos tener un puerto distinto para la aplicación al que en realidad estamos accediendo) y seleccione uno según la carga de trabajo que tengamos en el momento; esto es especialmente útil, ya que podemos tener replicada una aplicación y el servicio se encargaría de redirigir el tráfico de manera equitativa, distribuyendo la carga de trabajo entre todos los pods del mismo tipo (aunque también puede provocar errores intermitentes si el pod está dañado).
El uso de los services no acaba aquí, ya que existen distintos tipos de servicio que sirven a necesidades distintas. El que aparece en la imagen es de tipo ClusterIP y tiene la ventaja de que solo puede ser descubierto desde dentro del cluster, aislándolo del exterior. Otro ejemplo interesante sería el tipo NodePort que reserva un puerto en nuestros nodos y redirige el tráfico a nuestro servicio y que, en el caso de AWS, crea un balanceador de carga en el exterior, que permite que nuestras páginas puedan ser accedidas desde el exterior, al mismo tiempo que el balanceador se encarga de distribuir el tráfico entre los nodos y conseguir un mejor servicio.
Sin embargo esto acarrea un problema, y es la creación de un balanceador de carga por cada service NodePort que creamos. Esto no sería tan malo si no fuera porque se cobra por cada uno de ellos por separado, de tal manera que al tener varias páginas en el cluster podríamos tener un coste significativo en nuestro entorno; por lo que tenemos que buscar otra alternativa (Ingress + Nginx).

4.3.5 Ingress

Ingress es otro componente que se encarga del acceso externo al cluster, el cual funciona en conjunto con un balanceador de carga para acceder a los servicios que hemos definido previamente en el cluster (está pensado para servicios HTTP y HTTPS, por lo que necesitarás un servicio NodePort o LoadBalancer, si quieres alojar otro tipo de servicio).

Los campos a rellenar para el correcto funcionamiento de un ingress serían los dominios que queremos registrar junto con sus paths (/, /admin, /index, etc.) y unido a cada path el recurso service, al que apuntan junto al puerto en el que está alojado (en nuestro caso apuntaría al servicio de Varnish, ya que queremos que la página quede cacheada). Si queremos que conexiones sean de tipo HTTPS, tenemos que especificar el parámetro tls junto con los dominios que queremos asegurar y sus respectivos certificados, claves y entidades certificadoras de acceso (aprovechamos el uso de Secret para pasarles estos valores y en el campo de crt concatenamos en base64 la key y el certificado); y si no queremos que las conexiones vayan por HTTPS, simplemente omitimos todo el subconjunto de tls.

(*Referencia C16)

Con el uso de ingress podemos evitar la creación de múltiples balanceadores de carga y redirigir el tráfico entrante a través de un solo elemento; sin embargo hay un problema que debe ser resuelto antes de usar este elemento y es la instalación y uso de un controlador. Por defecto, no existe ningún tipo de controlador que se encargue de gestionar el direccionamiento de las peticiones, por lo que tenemos que instalar uno manualmente en nuestro cluster (varía según el tipo de proveedor cloud). Decidimos utilizar el controlador de nginx, que en esencia es un servidor con la aplicación nginx que recibe todas las peticiones y las redirige a los distintos servicios de nuestro cluster, según el dominio al que se esté solicitando acceso. Para instalar este controlador podemos usar la herramienta Helm o podemos sacar los ficheros YAML de la página https://kubernetes.github.io/ingress-nginx/deploy/#aws y lanzarlos con el comando kubectl, como si de cualquier otro componente normal se tratase (si investigamos los ficheros podemos observar que el motivo por el que solo hay un balanceador de carga es porque el deployment de nginx está expuesto al exterior con un service de tipo loadbalancer, que escucha en los puertos 80 y 443). Esta implementación requiere un pequeño ajuste, que obliga a que las comunicaciones del interior de nuestro cluster vayan por HTTP, ya que se produce terminación SSL y si forzamos el uso de HTTPS puede producirse un bucle de redirecciones, que constantemente alterna de peticiones HTTPS a HTTP y viceversa.

4.3.6 Cluster autoscaler

Gracias al uso de los Horizontalpodautoscalers somos capaces de replicar los pods que componen nuestro sistema, para satisfacer la demanda del servicio; sin embargo hay otro problema que debemos solucionar si queremos dotar a nuestro sistema de más capacidad de autoreplicación, que la replicación de pods no puede cubrir por sí sola. El problema es la asignación de pods a las instancias/nodos de nuestro cluster; ya que si un pod solicita cierta cantidad de recursos, pero no existen nodos con la capacidad suficiente para cubrirla, nos encontraremos con que nuestros pods se quedan en estado Pending, hasta que añadamos más instancias capaces de cubrir este incremento de la demanda.
Aunque este proceso se puede hacer manualmente, queremos que nuestro sistema sea capaz de realizar funciones de autoescalado de manera automática sin nuestra intervención, por lo que tenemos que volver a escalar una vez más en el nivel de funcionamiento de nuestro cluster y adentrarnos en la replicación a nivel de instancias de computación que forman un cluster EKS.

Para esta sección tenemos que volvernos un poco más específicos ya que el funcionamiento de esta parte puede variar según nuestro proveedor en la nube, por lo que tras un poco de investigación, hemos encontrado la manera de replicas las instancias de

EC2 de AWS con una combinación de políticas IAM y ficheros de configuración YAML en nuestro cluster.
Primero vamos a descargarnos un conjunto de recursos en formato yaml de la dirección https://eksworkshop.com/scaling/deploy_ca.files/cluster_autoscaler.yml. Si investigamos este fichero yaml, observamos que se están declarando un conjunto de permisos y roles que permiten la descripción, monitorización, creación y destrucción de recursos de Kubernetes (pods, services, etc.), al mismo tiempo que crea un deployment en el namespace kube-system, cuya función es lanzar un script que monitorizará el uso de recursos cada cierto tiempo en nuestro cluster y reorganizará la colocación de nuestros pods en las instancias según el uso de recursos, para aprovechar al máximo el número de instancias que disponemos, al mismo tiempo que solicitará más instancias EC2 del grupo que especifiquemos, en caso de que nuestro cluster no tenga la capacidad necesaria para alojar todas nuestras aplicaciones (esto nos permitirá reducir el número de instancias en aquellos momentos en los que no necesitemos usar tantos recursos, o el caso contrario, cuando exista un incremento en la demanda). Los parámetros que necesitamos modificar es: la región donde se están alojando nuestras instancias, al mismo tiempo que pasamos el nombre del grupo de autoescalado de nodos que creamos al principio (apartado Cluster) junto al número de nodos entre los que puede variar el número de instancias presentes en ese grupo. Tras esto tenemos que encontrar el rol creado para nuestro grupo de nodos en la sección de permisos en la consola de Amazon y asignarle una política que le permita ajustar el número de nodos deseados que queremos para ese grupo. Los permisos específicos son:
"autoscaling:DescribeAutoScalingGroups", "autoscaling:DescribeAutoScalingInstances", "autoscaling:SetDesiredCapacity", "autoscaling:TerminateInstanceInAutoScalingGroup"
Con este último paso lo único que queda es lanzar el yaml que hemos editado previamente con el comando kubectl y tendríamos lo necesario para el correcto funcionamiento del autoescalado de nuestras instancias (puedes inspeccionar su funcionamiento, sacando los logs del namespace kube-system y el pod de autoscaler que ha sido generado), que se realizará periódicamente.
Gracias a este último cambio la capacidad de autoreplicación automática ya no se limita solamente a los componentes individuales de nuestro sistema, si no que se expande a los entornos que alojan nuestros contenedores, llevando todas las ventajas del autoescalado de puntos previos a un nivel superior y permitiéndonos expandir y reducir nuestro sistema al completo cuando añadamos o borremos una página o el reduciendo enormemente el coste de nuestro cluster cuando las páginas tengan poco tráfico y sean reorganizadas para caber en un número inferior de nodos.

4.3.7 Helm, Grafana y Prometheus

Durante el desarrollo de algunos de nuestros componentes, he mencionado el uso de la herramienta Helm para la instalación de alguno de sus componentes. Helm es una herramienta para nuestro cluster de Kubernetes, que funciona a modo de instalador de aplicaciones complejas para nuestro entorno Kubernetes. La manera en la que funciona es instalando el componente llamado Tiller dentro de nuestro cluster, que se encarga de llevar un control sobre los Charts que se instalan en el entorno.
Un chart es, en esencia, los distintos proyectos formados por conjuntos de ficheros yaml que contienen recursos de Kubernetes, que en conjunto instalan una funcionalidad completa dentro del sistema. Por poner un ejemplo, el chart stable/mysql es un conjunto de recursos deployment, service, persistent volumen, secret y configmap, que crean una base de datos en nuestro cluster, sin tener que construir individualmente cada uno de los elementos necesarios para poner la base de datos en funcionamiento.

En esencia, siempre sería posible conseguir el mismo resultado lanzando individualmente todos y cada uno de los yaml que componen el sistema que queremos montar (siempre y cuando tengamos acceso a las imágenes que usan), pero gracias a la aplicación Helm podemos abstraer las partes más complejas de la configuración de algunos sistemas e incluso personalizar algunos aspectos de los mismos, pasando un fichero values.yaml, que se encuentra presente en casi todos los charts de Helm y que su configuración varía según las especificaciones del creador del chart (incluso puedes crearte tu propio chart desde cero).

Como parte final de nuestra monitorización del cluster, queremos añadir Grafana, una herramienta para el análisis y monitorización de métricas con elementos gráficos, junto con Prometheus, herramienta para almacenar métricas, a la que le pasaremos datos relacionados con el entorno Kubernetes, que luego serán pasados a Grafana.
Para el correcto funcionamiento de la herramienta, necesitaremos que Prometheus reciba las métricas de todos y cada uno de nuestros nodos, al mismo tiempo que son pasados a Grafana para poderlos visualizar de manera cómoda y encontrar las partes que más uso hacen de nuestros recursos. Esta implementación puede ser complicada, ya que hay que lanzar pods en todos los nodos del cluster (recordemos que se crean y destruyen automáticamente), que almacenen métricas relacionadas al funcionamiento de Kubernetes, que al mismo tiempo sean almacenadas por Prometheus y que después puedan ser recogidas por Grafana, para que puedan ser convertidas en gráficas que editaremos a nuestro gusto.
Como se puede ver, no es una configuración sencilla, pero vamos a aprovechar la potencia de Helm para crear las partes más complejas del sistema, para luego editarlas a nuestro gusto. Primero buscamos un chart que cumpla nuestras necesidades y, entre todos ellos, encontramos stable/prometheus-operator. Este chart es realmente complejo y se compone de una gran cantidad de componentes, que en conjunto trabajan para cumplir los requisitos que hemos definido: prometheus, node-exporter, kube-state-metrics, grafana y más. Sin embargo con el uso del comando helm install stable/prometheus-operator --namespace=<namespace> podemos tener todo lo necesario para tener Grafana funcionando en nuestro cluster (habría que utilizar el comando kubectl port-fordward con el pod de grafana para acceder al mismo).
Solo con el comando anterior tendríamos lo necesario para monitorizar los recursos que tenemos en nuestro cluster, sin embargo vamos a personalizarlo ligeramente para satisfacer algunas necesidades más específicas.
Para empezar, la idea de usar port-forward para acceder a Grafana solo sirve para el entorno de desarrollo, por lo que creamos un recurso Ingress para poder acceder desde el exterior, como si de una página web más se tratase; lo único que habría que hacer es coger la plantilla de Ingress definida previamente y crear otro recurso en el namespace que hemos definido con el comando anterior y hacer que apunte al servicio que ha creado la herramienta Helm por nosotros (por defecto, prometheus-operator-grafana en el puerto 80).
Además de las métricas de Kubernetes, nos gustaría ser capaces de recolectar algunas métricas de Cloudwatch, por lo que vamos a utilizar stable/prometheus-cloudwatch-exporter en el mismo namespace, para recuperar esas métricas de manera similar a node-exporter (crearemos uno para s3 y otro para cloudfront, modificando el fichero values.yaml y lanzándolos por separado con distintos nombres).
El problema con el apartado anterior se encuentra a la hora de pasar esas métricas a prometheus, pero para solucionarlo hemos investigado el código del chart de operator y hemos descubierto que para pasar las métricas utiliza el recurso ServiceMonitor, por lo que creando el nuestro personalizado, somos capaces de engañar al chart original para que

recupere las métricas, como si se tratasen de las de Kubernetes, duplicando uno de los existentes y haciendo que apunte a nuestros exportadores.

4.4 Entornos de testing

Antes de lanzar nuestro código a un entorno de producción es muy probable que queramos probar las modificaciones en un entorno de testing, donde podamos estar seguros de que pueda funcionar correctamente, antes de que el código llegue al público.
Lo primero que pensamos es en un entorno bare-hardware, ya que podemos probar nuestro código sin tener que recurrir a recursos fuera de nuestro entorno de desarrollo; pero el problema que tenemos con este método es que cuando queremos probar nuestro código antes de que llegue al entorno de producción, nos gustaría que se parezca lo más posible al entorno que tenemos en la nube.
Para mitigar este último factor, podemos empezar por lanzar nuestras imágenes en nuestro equipo, utilizando la propia herramienta de Docker y arrancando el contenedor en nuestra máquina, usando el comando docker run. Esta solución requiere ciertas modificaciones, como hacer que los contenedores apunten a recursos de nuestro local (ya sea la propia máquina donde alojamos el contenedor u otros contenedores arrancados en nuestro equipo) y también implica saltarse todo el entorno de Kubernetes; por lo que la configuración con la que arrancamos la imagen se parece poco a lo que habría en nuestro entorno real.

Para acercarnos un poco más a lo que sería un entorno real de Kubernetes, podemos usar la herramienta Minikube, ya que su función es crear un entorno Kubernetes con un solo nodo dentro de una máquina virtual, que estará alojada en nuestro equipo. Con esta herramienta, disponemos de acceso a las estructuras que se pueden generar con el uso de comando kubectl, por lo que seremos capaces de recrear casi todos los comportamientos que he ido explicando a lo largo de este documento.
Sin embargo, a pesar de lo similar que puede ser un entorno en Minikube, hay un problema que no puede ser solucionado y viene con el propio entorno de desarrollo, el cual se corresponde con el uso de recursos específicos que solo se encuentran disponibles, según el proveedor de servicios, en la nube que estemos utilizando.
Minikube ya viene con el defecto de solo tener un nodo donde alojar nuestros pods (lo cual deja el autoescalado de nodos y el selector de nodos sin uso dentro de nuestro entorno de testing), lo cual puede llevar a errores de configuración al desarrollar, pensando que todos los pods se alojarán en un mismo nodo; tampoco no es compatible con balanceadores de carga (y todas aquellas funcionalidades que dependan de un proveedor de servicios en la nube) y algunas de las funcionalidades que usamos, como el controlador nginx para nuestro ingress, vienen instaladas por defecto.
Todavía es posible realizar casi toda la funcionalidad necesaria de nuestro sistema en nuestro entorno bare-hardware si usamos Minikube, pero si queremos ir más lejos y acerarnos todavía más a nuestro entorno de producción, siempre podemos abandonar la idea de realizar pruebas en un entorno bare-hardware y pasar directamente al entrono de producción.
Al principio, la idea había sido descartada, porque pensábamos que al realizar modificaciones en el entorno de producción, provocaría fallos en las otras páginas; pero según fuimos avanzando en el desarrollo, se hizo evidente que el uso de namespaces podía solucionar todos nuestros problemas, ya que si usábamos namespaces distintos para nuestros entornos de testing, podríamos ser capaces de desplegar páginas sin que el resto de elementos del cluster fueran siquiera conscientes de su existencia, al mismo tiempo que creábamos un entorno que sería prácticamente idéntico al entorno que tenemos en producción

5 Integración, pruebas y resultados

Con el cluster en pleno funcionamiento podemos comprobar que muchas de las funcionalidades que buscábamos a la hora de asegurar nuestro cluster funcionan correctamente:

- Cuando el número de peticiones en la página se incrementa, el uso de CPU supera el umbral que definimos en el autoscaler y los pod se duplican para satisfacer la demanda.
- Los servicios distribuyen la carga que llega a los deployment haciendo que, una vez que se replican los pods, el tráfico se reparta entre ellos para aligerar la carga de trabajo en una sola instancia.
- Los nodos escalan automáticamente sin nuestra intervención, disminuyendo su número cuando el tráfico es menos intenso y ahorrando recursos en esas horas de menos uso (y también el caso contrario cuando necesitamos más recursos para satisfacer la demanda).
- Cada uno de los elementos de nuestra página web están aislados de tal manera que si uno cae, el resto no se ven afectados.
- Gracias al uso de contenedores y la tecnología de Kubernetes, kubectl vigila el estado de nuestros pods, de tal manera que si uno sufre un error crítico es capaz de, automáticamente, volver a un estado previo y reanudar su servicio.
- Gracias al uso de la herramienta Sentry somos capaces de recibir notificaciones cada vez que nuestro código sufre una excepción.
- Gracias a herramientas como Prometheus y Grafana podemos monitorizar el uso de nuestro cluster, para saber cómo optimizar la asignación de los recursos a nuestros elementos (y también identificar ataques cuando se produce un exceso de tráfico o cuando se produce un uso excesivo de recursos, debido a una mala configuración de la cache).
- Gracias a herramientas como Helm podemos instalar componentes complejos en nuestro sistema, que pueden ayudarnos a mejorar la funcionalidad de nuestras páginas (como la notificación por Slack cuando se caen nuestros spot-instances).
- Con el uso de frameworks, como Magento, abstraemos funcionalidad ante el programador y evitamos ataques más básicos, como inyección SQL.
- Gracias al fichero de acceso auth.yaml podemos editar fácilmente quién tiene acceso al cluster, por lo que limitando el acceso, rápidamente podemos evitar que el daño que se produce por el robo de cuentas con phising se extienda a nuestro cluster (solo el creador tiene acceso y puede dar acceso momentáneo a otros usuarios a través de ese fichero).

Como medida adicional, se lanzaron algunas pruebas con Apache Benchmark, con el cual queríamos comprobar el rendimiento de la página en comparación con el entorno antiguo. Para realizar estas pruebas, lanzamos 1000 peticiones con 100 peticiones concurrentes sobre páginas que solo cachean parte de su contenido, como son la página de categorías y de productos. Los resultados demostraron que (los primeros resultados corresponden a los del cluster), para la página de categorías, teníamos unas 42 peticiones por segundo frente a 46 peticiones por segundo, junto con un tiempo de 2.4 segundos frente a otro de 2.2 segundos; mientras que para la página de producto obteníamos 44-45 peticiones por segundo y unos tiempos prácticamente idénticos de 2.2 segundos.
Por los resultados, podemos observar que nuestro cluster es bastante similar a la velocidad normal del antiguo servidor, pero con una diferencia importante: aunque al principio

nuestro cluster estaba un poco por debajo del rendimiento antiguo, tras realizar unas pocas pruebas durante un tiempo, nuestros pods empezaron a replicarse debido al incremento constante de uso de CPU, y empezaron a alcanzar e incluso superar el rendimiento del antiguo servidor, lo cual tiene la ventaja de usar pocos recursos de nuestro cluster durante horas de poco uso y tras un breve tiempo de actividad se replica hasta alcanzar un rendimiento estable (aunque hay que reconocer que la página sufre una pequeña bajada de rendimiento hasta que se detecta la necesidad de duplicar los pods).

Alguna pruebas adicionales que pudimos realizar en el cluster una vez que teníamos Grafana y Prometheus instalados (y tras esperar un tiempo para reunir suficientes datos) fue identificar los recursos que necesitaban cada uno de los componentes de cada namespace. Podemos diferenciar nuestros contenedores en cpu-intensive y mem-intensive según el tipo de recurso que más necesitan y con los datos recopilados durante el tiempo que Prometheus esté recuperando métricas podemos descubrir cuáles son los recursos óptimos para cada componente de nuestra página y ajustarlos a los campos de request y limit de los yaml de nuestros deployments. Esto último es especialmente útil ya que nos permite minimizar los costes asignando a los pods a nodos que se ajustan a las necesidades específicas de sus aplicaciones; y limitar sus recursos para que Kubernetes pueda planificar de manera más eficiente la liberación de recursos y la expulsión de procesos del procesador según su necesidad.
Para entender esto mejor podemos usar el ejemplo del contenedor de Varnish. En el caso de la CPU pudimos observar que rara vez pasaba de los 100 milicores de uso llegando a estar por debajo de los 50 en momentos de poco uso; sin embargo fue en la memoria donde se pudo observar una de las ventajas de la limitación de recursos en Kubernetes. La limitación de recursos en Kubernetes evita que uno de los componentes de nuestra página crezca sin control, quitando recursos a otros componentes y llegando a saturar el sistema si no estuviese montando un entorno con capacidad de recuperación; esto se puede observar en la memoria de Varnish ya que al establecer un límite especialmente bajo para algunas páginas el porcentaje de uso asignado superaba el 100%, sin embargo no era expulsado ya que no había otro componente que necesitara los recursos en ese momento; pero cuando esto ocurría Kubernetes forzaba al pod a liberar recursos y dejar espacio para otros componentes, al mismo tiempo que le dejaba volver a crecer una vez que no fuera necesaria la utilización de recursos por otros componentes. Para mitigar las caídas de memoria pudimos estudiar el transcurso en el tiempo del uso de memoria y aumentamos sus recursos hasta conseguir una línea casi continua (hay que tener en cuenta limpiezas de cache, actualización de estáticos, tráfico en la página y su tamaño, etc.), pero gracias a este estudio pudimos comprobar la versatilidad de Kubernetes a la hora de gestionar los recursos de nuestro cluster.
Tras comprobar todos los componentes llegamos a la conclusión de que el servidor Apache entraría en el grupo de cpu-intensive, mientras que el resto son mem-intensive.

Figura 5-1: Uso de memoria de Varnish en distintas páginas

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Debido a nuestra nueva implementación contamos con entornos aislados que pueden ser modificados dinámicamente según el tráfico de nuestro sistema, que reduce el número de recursos en las horas de uso más bajas, al mismo tiempo que incrementa su capacidad cuando la necesidad de más capacidad para cumplir con la carga de trabajo sea necesaria. También, debido a la facilidad de crear y destruir componentes enteros del sistema, somos capaces de introducir nuevas funcionalidades en el cluster, sin experimentar tiempos de caída excesivos, al mismo tiempo de que somos capaces de volver a versiones anteriores de los componentes que sufran errores de desarrollo en cuestión de segundos y sin que nuestros clientes sufran las consecuencias de una larga pérdida del servicio.

Las ventajas no acaban aquí, sino que con los nuevos sistemas de monitorización con los que contamos somos capaces de estudiar el estado de nuestro sistema para ver el consumo de recursos de sus componentes individuales, al mismo tiempo de que somos informados cada vez que se produce un error en uno de nuestros componentes; además, gracias a la naturaleza efímera de los pods somos capaces de recuperarnos automáticamente de daños que, en otros tipos de sistemas, habrían provocado una caída importante del servicio hasta que el problema hubiese sido resuelto.

Todas estas características beneficiosas no vienen solo de los sistemas distribuidos sino que el uso de contenedores nos ayuda a crear entornos aislados desde cero, que dejan atrás aquellos tiempos en los que los componentes de un sistema dejaban de funcionar una vez que migrábamos de un entorno a otro (además de aprovechar el aislamiento de los contenedores, el uso de distintos espacios dentro de la misma máquina aumentan la capacidad de aislamiento de nuestras páginas, incrementando la seguridad de las mismas).

El número de ventajas que podemos sacar a este nuevo de entorno son innumerables y dentro de poco tiempo podrían establecer un nuevo estándar a la hora de de suministrar toda clase de servicios al público, que revolucionarán la manera en la que los programadores dan uso a sus sistemas distribuidos.

6.2 Trabajo futuro

Aunque la creación de un nuevo entorno seguro nos ofrece una cantidad enorme de ventajas respecto a sistemas tradicionales, hay que tener en cuenta un factor crítico a la hora de crear cualquier tipo de infraestructura, estoy hablando concretamente de los **costes** que supone mantener al mes nuestra nueva infraestructura.

Dependiendo del proveedor de servicios que utilices y de la naturaleza del sistema que estés migrando, el coste por mantener el cluster en funcionamiento y los sistemas auxiliares de vigilancia pueden provocar un incremento significativo en la factura a final de mes. Por poner como ejemplo el propio proveedor de AWS, las instancias de **EC2** generadas por demanda suponen un gasto significativo todos los meses, a cambio de una adquisición rápida y servicio constante, mientras que las mencionadas brevemente en nuestro proyecto como **spot-instances** son significativamente más baratas, pero corres más riesgo de perderlas debido al sistema de subasta y demanda de Amazon. Todo esto sin tener en cuenta de que Amazon tiene límites sobre el número de instancias (demanda)

que puedes solicitar, siendo necesario pedir un incremento del límite que se te concede según el uso de los servicios que hasta el momento has ido utilizando; lo que puede provocar que en un momento crítico de aumento de tráfico te quedes sin suficientes unidades de computación.

También el uso de sistemas de métricas de **Cloudwatch** supone otro gasto significativo, ya que Amazon te cobra por la solicitud periódica de ciertas métricas; además de cobrarte extra por monitorizar métricas de cierta naturaleza que pueden influir al balance de final de mes (por ejemplo, el número de peticiones a un bucket de S3 es una métrica extraordinaria que hay que solicitar).

A raíz de los problemas anteriores se podrían proponer un trabajo que estudie varias ***alternativas a la hora de elegir un proveedor de servicios*** (DigitalOcean, Azure, etc.) comparando sus puntos fuertes y débiles, haciendo hincapié en los ***costes*** necesarios para mantener nuestros sistemas, de tal manera que seamos capaces de elegir un entorno de producción que no solo sea económico sino que cumpla con todas las necesidades y controles de ***calidad*** para dar servicio a nuestros clientes (*un estudio de alternativas con relación calidad-precio*).

Referencias

[1] Margaret Rouse, Cameron McKenzie, “Contenerización de aplicación (contenerización de app), Mayo 2017”. https://searchdatacenter.techtarget.com/es/definicion/Contenerizacion-de-aplicacion-contenerizacion-de-app

[2] Israel Villafuerte, “Docker, qué es y cómo funciona la contenerización”, 4 Julio 2016 https://www.stackfire.com/docker-que-es-y-como-funciona-la-contenerizacion/

[3] “rkt, A security-minded, standards-based container engine” https://coreos.com/rkt/

[4] “Singularity Info” https://www.sylabs.io/singularity/

[5] Bill Doerrfeld, “5 Container Alternatives to Docker”, 22 Enero 2019 https://containerjournal.com/2019/01/22/5-container-alternatives-to-docker/

[6] Universidad Internacional de Valencia, “Sistemas distribuidos, características y clasificación”, 22 Febrero 2017 https://www.universidadviu.es/sistemas-distribuidos-caracteristicas-clasificacion/

[7] Kubernetes team, “What is Kubernetes?”, 20 Abril 2019 https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#what-does-kubernetes-mean-k8s

[8] Docker team, “Get Started, Part 4: Swarms”, https://docs.docker.com/get-started/part4/

[9] Marathon team, “Marathon, A container orchestration platform for Mesos and DC/OS” https://mesosphere.github.io/marathon/

[10] Twain Taylor, “Kubernetes alternatives: A look at Swarm, Marathon, Nomad, & Kontena”, 1 Diciembre 2017http://techgenix.com/kubernetes-alternatives/

[11] Microsoft Azure team, “¿Qué es un proveedor de servicios en la nube?” https://azure.microsoft.com/es-es/overview/what-is-a-cloud-provider/

[12] Joan Carles, “¿Qué son los servicios en la nube?”, 5 Febrero 2016https://geekland.eu/que-son-los-servicios-en-la-nube/

[13] Amazon team, “Acerca de AWS” https://aws.amazon.com/es/about-aws/

[14] Amazon team, “Productos en la nube” https://aws.amazon.com/es/products/?nc2=h_m1

[15] Microsoft Azure team, “¿Qué es Azure?” https://azure.microsoft.com/es-es/overview/what-is-azure/

[16] Microsoft Azure team, “Azure Kubernetes Service (AKS)” https://azure.microsoft.com/es-es/services/kubernetes-service/

[17] Digital Ocean team, “Digital Ocean” https://www.digitalocean.com/

Glosario

API	Application Programming Interface
AWS	Amazon Web Services
VM	Virtual Machine
k8s	Kubernetes
URI	Uniform resource identifier
ECR	Elastic Container Registry
VPC	Virtual Private Cloud
EKS	Elastic Container Service
IAM	Identity and Access Management
CDN	Content Delivery Network
CORS	Cross-origin resource sharing
EBS	Elastic Block Store

Anexos

A Manual de instalación

En esta sección vamos a describir el proceso de instalación de las distintas herramientas que hemos utilizado para la creación de nuestro entorno seguro. La instalación de estas herramientas se realizó en un entorno MAC pero (con algunas modificaciones) debería ser posible la instalación de estas herramientas en un entorno Linux. Las herramientas a instalar son las siguientes:

- **Docker**: La instalación de la aplicación de docker es realmente sencilla ya que solo es necesario descargarse la aplicación de Docker for MAC para que se instalen todos los componentes necesarios en nuestro sistema (docker + máquina virtual HyperKit), todo esto con solo crearse una cuenta en un página web. Sin embargo esta aplicación solo está disponible para MAC, por lo que si queremos tener nuestra aplicación en otro sistema, como por ejemplo Linux, será necesario instalarse la alternativa de Docker CE descargándote la GPG key de Docker (*curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -)* y luego instalando la aplicación con *sudo apt-get update && sudo apt-get install docker-ce docker-ce-cli containerd.io.*
- **Minikube**: Para la gestión de un entorno cluster en nuestro equipo local podemos utilizar la herramienta Minikube la cual se comportará de manera similar a lo que haría un cluster permitiéndonos crear entorno de testing en nuestro equipo. Para instalarlo solo tenemos que tener un Hipervisor instalado (si estás utilizando Docker for MAC ya vendrá con el de HiperKit, aunque también podrías utilizar el de VirtualBox) y lanzar en comando *brew cask install minikube* si estamos usando MAC (la alternativa para entornos Linux o para los que no usan brew sería descargarse los binarios y añadirlos a nuestro path /usr/local/bin). Una vez instalado basta con lanzarlo con *minikube start* y el flag --vm-driver=x apuntando al Hipervisor que tengamos instalado y el propio comando modificará nuestro fichero de configuración para que los comandos kubectl funcionen sobre él (puedes ver el cluster al que estás atacando usando *kubectl cluster-info*).
- **Kubectl**: La instalación del comando kubectl requiere de un solo comando si tenemos brew instalado en nuestro equipo, *brew install kubernetes-cli*. Para instalarlo sin el uso de brew (disponible en MAC) es posible utilizar otros tipos de gestión de paquetes, como por ejemplo snap, que con solo el comando *sudo snap install kubectl –classic* tendríamos ya instalado kubernetes. El siguiente paso sería definir el entorno donde se lanzan nuestros comandos, el cual se puede definir modificando el fichero ~/.kube/config con la configuración de nuestro cluster, pero normalmente la mejor alternativa suele ser utilizar comandos específicos de nuestro cluster para generar el fichero en vez de modificarlo nosotros a mano (minikube genera automáticamente el fichero al crear un entorno y para AmazonWebServices el comando *aws eks update-kubeconfig –name CLUSTERNAME* debería hacer lo mismo pero para interactuar con un cluster tipo EKS). Existe también un ejecutable para autocompletar nuestros comandos kubectl el cual se puede instalar de muchas maneras (para MAC hay que usar *brew install bash-completion@2*) pero para que funcione en nuestra terminal hay que ir al fichero ~/.bashrc y añadir (las tres primeras líneas son para MAC):

```
export BASH_COMPLETION_COMPAT_DIR=/usr/local/etc/bash_completion.d
```

```
[[ -r /usr/local/etc/profile.d/bash_completion.sh ]] && .
/usr/local/etc/profile.d/bash_completion.sh
echo 'source <(kubectl completion bash)'
```

Existen variaciones a la hora de instalar este último componente según el entorno que utilices pero puedes encontrar la más información al final de la segunda fuente de esta sección.

- **Helm**: Podemos instalar este comando con la herramienta brew, *brew install kubernetes-helm*. La funcionalidad de este comando es instalar charts en nuestro cluster, que son conjuntos de recursos ya preparados que se generaran en nuestro cluster y nos proporcionarán una estructura prefabricada que podemos editar con un fichero yaml. Esta herramienta es completamente opcional ya que siempre podemos crearnos nosotros los ficheros y lanzarlos con el comando apply de kubelet, aunque gracias a esta herramienta podemos crear estructuras muy complejas con el uso de un solo comando. Para empezar a trabajar con helm hay que tener acceso a un cluster y lanzar *helm init*, esto instalará tiller en nuestro sistema (el cual se encarga de crear los componentes en el sistema) y nos permitirá instalar los charts dentro de la branch stable de helm (se pueden añadir otras, como incubator, con el uso de *helm add <repo>*).

- **Aws-cli y authenticator**: Necesitaremos estas herramientas para simplificar los cambios de entorno cada vez que queramos cambiar de cluster (aunque podría hacerse modificando el fichero $HOME/.kube/config) y para ser capaz de identificarnos cada vez que queramos acceder al cluster. Para instalar aws-cli necesitamos Python instalado en nuestro sistema y con el uso del comando pip podemos realizar la instalación: pip3 install awscli –upgrade –user. Si no queremos usar pip siempre podemos descargarnos con curl el instalador, descomprimirlo y ejecutarlo poniendo las rutas a nuestros directorios para que se creen enlaces simbólicos y se ejecuten como comandos normales. Un ejemplo para MAC sería:

```
curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"
unzip awscli-bundle.zip
sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
```

Para instalar autheticator y poder conectarnos al cluster necesitaremos seguir un procedimiento similar al anterior; usando un instalador y redirigiendo el ejecutable a nuestra terminal:

```
#Cambia en la URL darwin por linux o windows según tu sistema operativo (en Windows hay que añadir .exe al final)
curl -o aws-iam-authenticator https://amazon-eks.s3-us-west-2.amazonaws.com/1.12.7/2019-03-27/bin/darwin/amd64/aws-iam-authenticator
chmod +x ./aws-iam-authenticator

#mkdir solo es necesario si no existe el directorio de ejecutables (improbable)
mkdir $HOME/bin && cp ./aws-iam-authenticator $HOME/bin/aws-iam-authenticator && export PATH=$HOME/bin:$PATH
```

```
#Añadimos el PATH a nuestro ejecutable para poder arrancarlo desde terminal
echo 'export PATH=$HOME/bin:$PATH' >> ~/.bash_profile
#En Linux sería
#echo 'export PATH=$HOME/bin:$PATH' >> ~/.bashrc
```

Guías online:

https://kubernetes.io/docs/tasks/tools/install-minikube/
https://kubernetes.io/docs/tasks/tools/install-kubectl/
https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
https://docs.aws.amazon.com/cli/latest/userguide/install-macos.html
https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html
https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-docker-ce
https://docs.docker.com/docker-for-mac/install/

B Manual del programador

Para ser capaz de desplegar aplicaciones en nuestro cluster en kubernetes debemos dominar los comandos de docker y kubectl. A continuación dejo una lista de los comandos que el programador necesitará (junto con algunas variaciones útiles) según el nivel de la aplicación donde se encuentre.

B.1 Docker

Muchas de las opciones que verás a continuación se pueden encontrar en https://docs.docker.com/engine/reference/commandline/docker/

- docker build –t <nombre>:<tag> <directorio>:
  Este comando es la base para la creación de nuestras imágenes. Para su correcto funcionamiento tenemos que especificar el nombre que le pondremos a la imagen junto con la ruta a la localización de nuestro fichero Dockerfile que contiene las instrucciones para generar la imagen. Algunos de los flags que necesitaremos serán:
  - --build-arg <nombre>=<valor>: Este flag nos permite asignar valores a las variables de nuestro Dockerfile que se hayan especificado con instrucciones ARG
  - --no-cache: Evita que docker use la cache para instrucciones intermedias de nuestro Dockerfile. Se suele utilizar para debugear nuestra instalación o para comparar tiempos a la hora de construir una imagen por primera vez.
- docker run <nombre_imagen>:<tag> -p <puerto_salida>:<puerto_imagen>:
  Con este comando arrancamos una imagen en nuestro contenedor y mientras el proceso principal de nuestra imagen no se cierre debería mantener nuestro contenedor funcionando. Con el flag p especificamos el puerto de salida a nuestro localhost para acceder a la imagen (se pueden especificar más de uno). El tag es opcional (por defecto va a latest) y nos permite arrancar imágenes con el tag que se creó. Algunos flags útiles:
  - –d: lanzarlo en background para dejar nuestra terminal libre.
  - --privileged: otorga privilegios a nuestro contenedor y le permite acceder a distintos devices y montar volúmenes en nuestro contenedor (con --device puedes especificarlos uno a uno).
  - –e <nombre>=<valor>: asigna un valor a una variable de entorno definida en el Dockerfile.
- docker exec –it <containerID> <command>:
  Nos permite lanzar comandos en contenedores que estén en funcionamiento. Se puede encontrar el id del contenedor si lanzamos el comando *docker ps*. El flag –it es una combinación de los flags i y t que nos permite interactuar con la imagen redirigiendo su STDIN, esto es especialmente útil si lanzamos un /bin/bash ya que nos permite arrancar una terminal interactiva en la imagen (esto también se puede hacer en un run arrancando un nuevo contenedor).
- docker image ls –a / docker container ls –a:
  Listar todas las imágenes/contenedores
- docker tag <imagen_local>:<tag> <repositorio>:<tag> / docker push <repositorio>:<tag>:
  Enlazar una imagen local con la de un repositorio y subirla

B.2 Kubernetes

- kubectl apply –f fichero.yaml:
  Con este comando creamos elementos dentro del cluster o actualizamos los ya existentes. Hay muchas maneras de generar elementos del cluster o al, ya sea usando el comando create seguido con del elemento que queremos generar o utilizando rolling updates en los que podemos usar (por ejemplo) el comando set para cambiar la imagen de unos pods ya arrancados seguido de un rollout undo para deshacerlo si nos hemos equivocado; pero tras estar trabajando un tiempo con el cluster creo que es más cómodo y seguro generarse un fichero yaml con los elementos que queremos generar ya que nos permite editar con gran facilidad cualquier tipo de componente dejando un registro de lo que hemos lanzado sin necesidad de tener que lanzar otro comando para inspeccionar los resultados (y ahorras tiempo buscando el comando el comando en tu historial si te equivocaste).
- kubectl delete –f fichero.yaml
  Este comando es la función inversa del apply, borrando cualquier elemento cuya clase y nombre coincidan con los del cluster. Como versión alternativa en vez de un fichero se puede especificar el tipo de recurso (pod, service, hpa, etc.) seguido de su nombre y borrarlo y si esta acción se realiza sobre un namespace se borra, no solo el namespace, sino todos los elementos que contenga (si el elemento está dentro de un namespace que no sea default tendrás que especificarlo con el flag -n).
- kubectl get <recurso> <nombre>
  Con este comando recuperamos los elementos que se encuentran en nuestro cluster. Si especificamos solo el tipo de recurso obtenemos todos los elementos de este tipo en el namespace default (con –n buscas en otro namespace y con --all-namespaces sacas todos). Al especificar el nombre del elemento podemos usar el flag –o para sacarlo en formato yaml o json y ver los detalles con los que se crearon.
- kubectl describe <recurso> <nombre>
  Se usa para describir elementos de nuestro sistema, dándonos más información de la que conseguiríamos con un get. Podemos sacar el estado del pod y su configuración actual permitiéndonos obtener información útil a la hora de debugear o ver los recursos que utiliza.
- kubectl logs <nombre_pod>
  Con este comando sacamos la salida del proceso principal que mantiene el pod en funcionamiento (se usa para debugear o monitorizar los contenedores dentro del pod). Con el flag --follow podemos seguir su salida en tiempo real y con el flag --previous podemos obtener la salida del pod antes de que se reiniciara (se puede utilizar para averiguar el motivo por el cual el contenedor dejó de funcionar).
- kubectl cp <namespace>/<pod>:/<filepath> <filepath2>
  Este comando se utiliza para copiar contenido de un pod a nuestro local y viceversa.
- kubectl explain <resource>
  Con este comando obtenemos los campos para una plantilla de un fichero yaml para crear el recurso especificado. También es posible describir un campo específico si ponemos un punto seguido del campo. Útil para describir las características que se pueden configurar a la hora de crear un elemento en el cluster.

C Referencias de código

C.1 Base Magento

```
FROM php:7.0-apache
ENV CURRENT_PROJECT_LOCATION /var/www/html/
ENV CURRENT_PROJECT_CONF /project/conf/

RUN mkdir -p ${CURRENT_PROJECT_CONF}
WORKDIR ${CURRENT_PROJECT_LOCATION}
RUN apt-get update && apt-get install -y g++ libicu-dev libfreetype6-dev libjpeg62-turbo-dev libmcrypt-
dev libpng-dev libjpeg-dev curl unzip sed zlib1g-dev sudo libxslt-dev
libyaml-dev python cron python-pip ruby-full \
   && docker-php-ext-configure gd --enable-gd-native-ttf --with-freetype-dir=/usr/include/freetype2 --with-
png-dir=/usr/include --with-jpeg-dir=/usr/include
   && docker-php-ext-install iconv bcmath mcrypt mbstring gd zip mysqli intl xsl pdo_mysql soap calendar
exif gettext pcntl sockets wddx opcache

RUN mv /usr/local/etc/php/php.ini-production /usr/local/etc/php/php.ini
RUN sed -i '/memory_limit = 128M/c\memory_limit = 2G' /usr/local/etc/php/php.ini \
   && sed -i '/;opcache.enable=0/c\opcache.enable=1' /usr/local/etc/php/php.ini \
   && sed -i '/; max_input_vars = 1000/c\max_input_vars = 100000' /usr/local/etc/php/php.ini \
   && sed -i '/;opcache.memory_consumption=64/c\opcache.memory_consumption=128'
/usr/local/etc/php/php.ini \
   && sed -i '/;opcache.interned_strings_buffer=4/c\opcache.interned_strings_buffer=8'
/usr/local/etc/php/php.ini \
   && sed -i '/;opcache.max_accelerated_files=2000/c\opcache.max_accelerated_files=4000'
/usr/local/etc/php/php.ini \
   && sed -i '/;opcache.revalidate_freq=2/c\opcache.revalidate_freq=60' /usr/local/etc/php/php.ini \
   && sed -i '/;opcache.fast_shutdown=0/c\opcache.fast_shutdown=1' /usr/local/etc/php/php.ini \
   && sed -i '/;opcache.enable_cli=0/c\opcache.enable_cli=1' /usr/local/etc/php/php.ini \
   && sed -i '/;opcache.validate_timestamps=1/c\opcache.validate_timestamps=0' /usr/local/etc/php/php.ini
\
   && sed -i '/; opcache.validate_root=0/aopcache.file_update_protection=0' /usr/local/etc/php/php.ini
RUN apt-get update -qq && apt-get install -y gnupg
RUN curl -sL https://deb.nodesource.com/setup_10.x | sudo bash -
RUN apt-get update -qq && apt-get install -y nodejs
RUN npm install --global gulp-cli
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer

WORKDIR /etc/apache2/mods-enabled/
RUN sudo ln -s ../mods-available/headers.load headers.load
RUN sudo a2enmod ssl
RUN sudo a2enmod proxy
RUN sudo a2enmod proxy_balancer
RUN sudo a2enmod proxy_http
RUN apt-get update -qq && apt-get install -y build-essential libcurl4-openssl-dev   libxml2-dev mime-
support libfuse-dev automake libtool wget tar pkg-config libssl1.0-dev       fuse git
RUN git clone https://github.com/s3fs-fuse/s3fs-fuse.git && cd s3fs-fuse && ./autogen.sh && ./configure
&& make && sudo make install && cd .. && rm -rf srfs-fuse/
RUN mkdir -p /usr/local/bin
RUN cat /usr/local/etc/php/php.ini
RUN php -me && php -i

WORKDIR ${CURRENT_PROJECT_LOCATION}
EXPOSE 80
EXPOSE 443
```

C.2 Partir de imagen base

```
FROM basic
ARG MAGENTO_VERSION
ADD --chown=www-data:www-data auth.json /var/www/.composer/auth.json
RUN chown www-data:www-data /var/www/.composer
RUN sudo -u www-data composer create-project --repository-url=https://repo.magento.com/
magento/project-community-edition=$MAGENTO_VERSION .
COPY --chown=www-data:www-data cron.sh /var/www/html/cron.sh
RUN chmod +x /var/www/html/cron.sh
RUN mkdir /scripts
COPY cronjobs /scripts
RUN find /scripts -type f -exec chmod +x {} \;
COPY cronjob /etc/cron.d
```

C.3 Imagen Varnish

```
FROM debian
RUN apt-get update && apt-get install -y varnish
RUN varnishd -V
ADD conf/default.vcl /etc/varnish/default.vcl
ADD conf/varnish /etc/default/varnish
RUN mkdir -p /usr/local/bin
ADD varnish.sh /usr/local/bin/varnish.sh
RUN chmod u+x /usr/local/bin/varnish.sh
ENTRYPOINT ["/usr/local/bin/varnish.sh"]
EXPOSE 80
```

C.4 Comando iniciar Varnish

```
exec /usr/sbin/varnishd -F -P /run/varnishd.pid -a :80 -f /etc/varnish/default.vcl -S /etc/varnish/secret -s
malloc,1430m
```

C.5 Backend Varnish

```
backend default {
   .host = "project-service";
   .port = "80";
}
```

```
acl purge {
   "127.0.0.1";
   "172.31.0.0/16";
}
```

C.6 Filtrado Varnish

```
if (
(req.url ~ "(%20|%2F)(s|S)(e|E)(l|L)(e|E)(c|C)(t|T) (%20|%2F)") ||
(req.url ~ "(%20|%2F)(u|U)(n|N)(i|I)(o|O)(n|N) (%20|%2F)") ||
(req.url ~ "(%20|%2F)(s|S)(l|L)(e|E) (e|E)(p|P) (%20|%2F)"))
{
      return(synth(403, "Acceso denegado"));
}
```

C.7 Master-slave MYSQL

```
#Para la imagen master
echo "CREATE USER
'$MYSQL_REPLICATION_USER'@'%'
```

```
#Para la imagen slave
echo "STOP SLAVE;" | "${mysql[@]}"
echo "CHANGE MASTER TO
```

```
IDENTIFIED BY
'$MYSQL_REPLICATION_PASSWORD' ;" |
"${mysql[@]}"
echo "GRANT REPLICATION SLAVE ON *.*
TO '$MYSQL_REPLICATION_USER'@'%'
IDENTIFIED BY
'$MYSQL_REPLICATION_PASSWORD' ;" |
"${mysql[@]}"
echo 'FLUSH PRIVILEGES ;' | "${mysql[@]}"
```

```
master_host='$MYSQL_MASTER_SERVICE_HOST'
, master_user='$MYSQL_REPLICATION_USER',
master_password='$MYSQL_REPLICATION_PASS
WORD' ;" | "${mysql[@]}"
echo "START SLAVE;" | "${mysql[@]}"
```

C.8 Montar Bucket

```
s3fs $${BUCKET}:/$${PATH} ${CURRENT_PROJECT_LOCATION}$${LOCAL} -o $${OPTIONS}
```

C.9 Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: $${NAME}-deployment
  namespace: $${NAMESPACE}
  labels:
    app: $${NAME}
spec:
  replicas: $${REPLICAS}
  selector:
    matchLabels:
      app: $${NAME}
  template:
    metadata:
      labels:
        app: $${NAME}
    spec:
      nodeSelector:
        ntype: cpu-OP
      containers:
        - name: $${NAME}
          image: $${IMAGE}:$${TAG}
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: "$${CPUR}"
              memory: "$${MEMR}"
            limits:
              cpu: "$${CPUL}"
              memory: "$${MEML}"
          securityContext:
            privileged: true
          env:
          - name: PRE_COMPILE
            value: "false"
          - name: LAUNCH_COMMAND
            value: "false"
          - name: COMMAND
            value: "php bin/magento maintenance:enable"
          - name: ENABLED_S3
            value: "$${ENABLED_S3}"
```

C.10 Volúmenes con secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: $${NAME}-mysql-secret
  namespace: $${NAMESPACE}
type: Opaque
data:
  username: $${USER} (tiene que venir en base 64)
  password: $${PASS} (tiene que venir en base 64)
```

```
volumeMounts:
        - name: $${NAME}-ebs-volume
          mountPath: /var/lib/mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: $${NAME}-mysql-secret
              key: password
```

C.11 Usar volumen ya existente

```
volumes:
- name: $${NAME}-ebs-volume
  awsElasticBlockStore:
    volumeID: $${VOLUMEID}
    fsType: ext4
```

C.12 Cronjob con uso de args

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: $${NAME}-cronjob-$${INDEX}
  namespace: $${NAMESPACE}
spec:
  schedule: "$${TIME}"
  concurrencyPolicy: Forbid
  suspend: false
  successfulJobsHistoryLimit: 1
  failedJobsHistoryLimit: 1
  startingDeadlineSeconds: 3600
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          nodeSelector:
            ntype: cpu-OP
          containers:
          - name: $${NAME}
            image: $${IMAGE}:$${TAG}
            securityContext:
              privileged: true
            resources:
              requests:
                cpu: "$${CPUR}"
                memory: "$${MEMR}"
              limits:
                cpu: "$${CPUL}"
                memory: "$${MEML}"
            env:
            - name: PRE_COMPILE
              value: "false"
```

```
        - name: LAUNCH_COMMAND
          value: "false"
        - name: COMMAND
          value: "php bin/magento maintenance:enable"
        - name: ENABLED_S3
          value: "$${ENABLED_S3}"
        args:
          -    echo
          -    “Hello world”
```

C.13 Entrypoint con Sentry

```
#!/bin/sh
export SENTRY_DSN=”${DIRECCION}”
if [ "$LAUNCH_COMMAND" = "true" ]; then
  exec "$@"
else
  mkdir /${FOLDER}
  s3fs ${BUCKET}:/${FOLDER} /${FOLDER} -o ${OPTIONS}
  mountpoint /${FOLDER}
  if [ $? -ne 0 ]; then
    sentry-cli send-event -m "Fallo al hacer backup para ${BUCKET} ${FOLDER}" --no-environ
    exit 1
  fi
  sqldate=$(date '+%Y_%m_%d')
  mysqldump -h ${HOST} -u ${USER} -p${PASS} ${DATABASE} > /$sqldate.sql
  if [ $? -ne 0 ]; then
    sentry-cli send-event -m "Fallo al hacer backup para ${BUCKET} ${FOLDER}" --no-environ
    exit 1
  fi
  mkdir -p /${FOLDER}
  zip /${FOLDER}/$sqldate.zip /$sqldate.sql
  if [ $? -ne 0 ]; then
    sentry-cli send-event -m "Fallo al copiar el backup a ${BUCKET} ${FOLDER}" --no-environ
    exit 1
  fi
  echo "BACKUP CREADO"
fi
```

C.14 Pod autoscaler

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: $${NAME}-autoscaler
  namespace: $${NAMESPACE}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: $${NAME}-deployment
  minReplicas: $${MIN_REPLICAS}
  maxReplicas: $${MAX_REPLICAS}
  targetCPUUtilizationPercentage: $${targetCPUUtilizationPercentage}
```

C.15 Service

```
apiVersion: v1
kind: Service
metadata:
  name: project-service
  namespace: $${NAMESPACE}
  labels:
    app: $${NAME}
spec:
  type: ClusterIP
  selector:
    app: $${NAME}
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
```

C.16 Ingress son SSL

```
apiVersion: v1
kind: Secret
metadata:
  name: $${NAME}-ingress-tls
  namespace: $${NAMESPACE}
type: Opaque
data:
  tls.crt: $${encoded_cert}
  tls.key: $${encoded_key}
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: $${NAME}-ingress
  namespace: $${NAMESPACE}
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - $${DOMINIO}
    secretName: $${NAME}-ingress-tls
  rules:
  - host: $${DOMINIO}
    http:
      paths:
      - path: /
        backend:
          serviceName: $${SERVICE_NAME}
          servicePort: 80
```

C.17 Permisos IAM ejemplo

```
{
  “Version”: “2019-04-05”,
  “Statement”: [
    {
      “Effect”: “Allow”,
```

```
      "Action": [
        "s3:ListAllMyBuckets"
       ],
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
       ],
      "Resource": "arn:aws:s3:::bucket1"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject"
       ],
      "Resource": "arn:aws:s3:::bucket1/*"
    }
  ]
}
```